Algorithms for White-box Obfuscation Using Randomized Subcircuit Selection and Replacement

THESIS

Kenneth E. Norman, Major, USAF

AFIT/GCS/ENG/08-17

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCS/ENG/08-17

ALGORITHMS FOR WHITE-BOX OBFUSCATION USING RANDOMIZED SUBCIRCUIT SELECTION AND REPLACEMENT

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

Kenneth E. Norman, B.E.E., M.S.Eng.Mgt.
Major, USAF

27 March 2008

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCS/ENG/08-17

Algorithms for White-box Obfuscation Using Randomized Subcircuit Selection and Replacement

Kenneth E. Norman, B.E.E., M.S.Eng.Mgt.
Major, USAF

Approved:

/signed/ 27 Feb 2008

Lt Col J. Todd McDonald, Ph.D. (Chairman) Date

/signed/ 27 Feb 2008

Dr. Yong C. Kim (Member) Date

/signed/ 27 Feb 2008

Lt Col Stuart H. Kurkowski, Ph.D. (Member) Date

AFIT/GCS/ENG/08-17

Abstract

Software protection remains an active research area with the goal of preventing adversarial software exploitation such as reverse engineering, tampering, and piracy. Heuristic obfuscation techniques lack strong theoretical underpinnings while current theoretical research highlights the impossibility of creating general, efficient, and information theoretically secure obfuscators. In this research, we consider a bridge between these two worlds by examining obfuscators based on the Random Program Model (RPM). Such a model envisions the use of program encryption techniques which change the black-box (semantic) and white-box (structural) representations of underlying programs.

In this thesis we explore the possibilities for white-box transformation. Under an RPM formulation, if an adversary cannot distinguish an original program from either its obfuscated version (whose black-box behavior has been strategically altered) or a randomly generated program of comparable size, then the white-box intent of the original program has been sufficiently protected. One proposed method of creating such random indistinguishability is by choosing (at random) a program from a size-bounded set of all semantically equivalent possibilities.

Since full enumeration of reasonably sized programs is not possible, in this work we focus on obfuscators which introduce random white-box structural variation based on iterative selection and replacement. We design and develop an obfuscation framework for programmatic logic expressed as combinatorial Boolean circuits and compare six unique approaches for sub-circuit selection. We analyze the relative behavior of random and guided-random sub-circuit selection algorithms while showing their utility in producing random white-box structural variation.

Acknowledgements

To my wife and son: Thank you for your love and support. My success is equally yours, and for your sacrifices, I owe you more than I can ever repay. I love you both very much.

Professionally, I owe a debt of gratitude to my thesis advisor, Lt Col Todd McDonald, and my research partner, Capt Moses James. As an electrical engineer in a computer science program, I know I taxed their patience with my many questions. Thank you.

Kenneth E. Norman

Table of Contents

Page

Abstract iv

Acknowledgements v

List of Figures viii

List of Tables x

I. Introduction 1
  1.1 Problem area 1
    1.1.1 Motivating scenario 1
    1.1.2 Context 2
  1.2 Research objectives 4

II. Literature Review 5
  2.1 What is obfuscation? 5
    2.1.1 Preliminary definitions 5
    2.1.2 Classifications of obfuscation 5
    2.1.3 Theoretical definitions 6
      2.1.3.1 Virtual Black Box Obfuscation 7
      2.1.3.2 Indistinguishability Obfuscation 8
      2.1.3.3 Best-Possible Obfuscation 9
    2.1.4 Practical applications 9
  2.2 Shortfalls of current theoretical work 10
  2.3 Random Program Security Model 11
    2.3.1 Program encryption 12
    2.3.2 Intent protection 14

III. Methodology 15
  3.1 Notation 15
  3.2 Assumptions 15
    3.2.1 Programs represented as circuits 15
      3.2.1.1 Combinational circuits 16
      3.2.1.2 Directed acyclic multi-graphs 18
    3.2.2 Iterative randomization 21
    3.2.3 Circuit library exists 21
  3.3 Obfuscation toolkit 23
    3.3.1 CORGI: the circuit randomizer 23

Page
3.3.1.1 Development environment 23
3.3.1.2 Subcircuit selection and replacement . . 24
3.3.2 CXL: the circuit library 25
3.4 Empirical Approach 26
3.4.1 Key concepts 26
3.4.2 Properties of obfuscated circuits 28
3.4.3 White-box obfuscation algorithms 30

IV. Results 32
4.1 Overview 32
4.2 Limitations 33
4.2.1 Smart strategies 33
4.2.2 Introduced cycles 34
4.3 Analysis of subcircuit selection algorithms 35
4.3.1 Common functions 36
4.3.2 `RandomSingleGate` 36
4.3.3 `RandomTwoGates` 38
4.3.4 `RandomLevelTwoGates` 45
4.3.5 `FixedLevelTwoGates` 47
4.3.6 `LargestLevelTwoGates` 49
4.3.7 `OutputLevelTwoGates` 50
4.4 Runtime performance analysis 52

V. Conclusions 67
5.1 Contributions 67
5.2 Future work 68

Appendix A. CORGI software 70
A.1 CORGI architecture 70
A.1.1 Functionality 70
A.1.1.1 JGraphT 70
A.2 Non-selection algorithms 70
A.3 Selection algorithm behavior 74
A.4 Selection algorithm results 78
A.4.1 C17 with all algorithms 78
A.4.2 C880 with `OutputLevelTwoGates` 78

Bibliography 84

Vita 86

Index Index-1

List of Figures

Figure Page

1.1 Program Encryption 3
2.1 The Random Program Model 12
2.2 RPM obfuscation 13
2.3 Black box obfuscated program 13
3.1 The Random Program Model 17
3.2 ISCAS Benchmark Circuit C17 18
3.3 Graph examples 19
3.4 Iterative randomization 22
3.5 Circuit hierarchy example 28
3.6 Example histogram 29
4.1 Improper subcircuit selection creates cycles 35
4.2 Introduced control flow in ISCAS C17 42
4.3 Diffusion of replacements in ISCAS C17 43
4.4 How a replacement subcircuit creates a new control flow 44
4.5 Runtime data for `R1G` on C17 55
4.6 Runtime data for `R1G` on C880 56
4.7 Runtime data for `R2G` on C17 57
4.8 Runtime data for `R2G` on C880 58
4.9 Runtime data for `RL2G` on C17 59
4.10 Runtime data for `RL2G` on C880 60
4.11 Runtime data for `FL2G` on C17 61
4.12 Runtime data for `FL2G` on C880 62
4.13 Runtime data for `LL2G` on C17 63
4.14 Runtime data for `LL2G` on C880 64
4.15 Runtime data for `OL2G` on C17 65

Figure Page

4.16 Runtime data for `OL2G` on C880 66

A.1 CORGI UML class diagram 71

A.2 Behavior data for all six selection algorithms 75

A.3 Chart of behavior data: circuit height 76

A.4 Chart of behavior data: circuit width 77

A.5 Sample results of R1G and OL2G algorithms 79

A.6 Sample results of R2G and FL2G algorithms 80

A.7 Sample results of RL2G and LL2G algorithms 81

A.8 Sample result 1 of OL2G applied to C880 81

A.9 Sample result 2 of OL2G applied to C880 82

A.10 Sample result 3 of OL2G applied to C880 83

List of Tables

Table Page

3.1 Notation for the Random Program Model 16

3.2 Features and benefits of JGraphT 24

3.3 Candidate circuit properties 29

3.4 Candidate subcircuit selection algorithms 30

4.1 Summary of runtime data 53

ALGORITHMS FOR WHITE-BOX OBFUSCATION
USING RANDOMIZED
SUBCIRCUIT SELECTION AND REPLACEMENT

I. Introduction

Across the Department of Defense, it is increasingly difficult to find a weapon systems which does not rely upon software to perform its intended function. The United States Air Force in particular is reliant on software across every facet of its mission: air, space, and cyberspace. The ubiquity of software-based systems, and the interconnectedness of such systems, demands we protect them from our adversaries' prying eyes. In many cases, physical security is sufficient to thwart anyone who seeks to gain access to our systems. When physical security fails to protect our critical software, we must turn to alternate means. One such alternative is software obfuscation.

1.1 Problem area

Software obfuscation is not a new concept, but neither is it a well-defined discipline in practice. The concept of software obfuscation is in many ways the unraveling of sound development principles. The objective in software engineering is to produce systems which are defect-free, modular, maintainable, and extensible. A well-engineered system will function as efficiently as possible, and perform the job the user expects, in the manner he expects it. The objective in software obfuscation is to produce highly coupled, difficult-to-understand, complex systems which, nevertheless, perform the job the user expects, in the manner he expects it (though perhaps with less efficiency by comparison).

1.1.1 Motivating scenario. In early 2001, the world watched as the US and China found themselves at odds after what became known as the Hainan Island incident. In brief, a US EP-3 reconnaissance plane and a Chinese Shenyang J-8 collided, and the EP-3 was forced to make an emergency landing on Hainan Island off the south coast of China. According to a 2 April 2001 UPI press release [11],

> “[t]he EP-3 could not have landed in a better place for China or a worse one for U.S. military intelligence. Hainan island is host to one of China’s largest electronic signals intelligence complexes and is manned by experts who can glean critical information on the aircraft’s capabilities if they gain access to the Navy’s EP-3” . . . Pentagon sources said.

The crew was held hostage for 12 days before being released. The plane, however, remained on Hainan Island for a total of 94 days, during which time China had unfettered access to the equipment on board. If the EP-3 crew was unable to entirely destroy all information storage devices (and the software they contain) before they landed, then the Chinese had ample opportunity to learn about US collection methods and targets of interest during the time the plane was in their control. Even if their examination would have taken more than 94 days, it would have been easy enough to copy the code (from undamaged equipment) and analyze it *after* they returned the aircraft to US custody.

1.1.2 Context. This research augments earlier work initiated by Lt Col Todd McDonald for his doctorate degree. In his dissertation, McDonald described software obfuscation as protecting program intent [12]. The concept of *intent protection* stands in contrast to traditional definitions of obfuscation, all of which require that a program’s functionality remain unchanged (without regards to some acceptable degradation of time and/or space efficiency). Instead, McDonald takes inspiration from the field of cryptography and likens intent protection to data encryption. The idea is to transform a program in two ways—structurally *and* functionally. If functionality (that is, input/output behavior) must change, then it must also be possible to recover the original behavior (see Figure 1.1). McDonald further requires that an intent protected program be indistinguishable from any other program, selected ran-

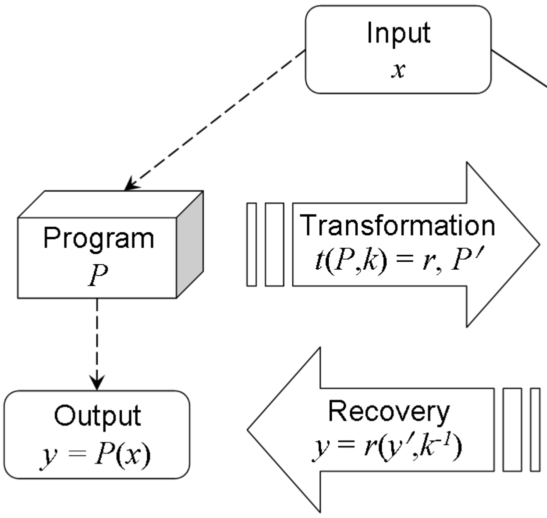


Figure 1.1: Program encryption under the Random Program Model

domly, which has a similar number of inputs, outputs, and is of similar size. This he calls the Random Program Model (RPM).

The difficult question is how to devise a random selection schema. Clearly, for any but the most basic of programs, software can be written in almost limitless ways to accomplish the same function. If the set is impossible (or at least infeasible) to create, an alternate means of "selection" is required.

Rather than attempt to enumerate entire sets of programs, then select a replacement *in toto*, we consider an alternate approach of iterative randomization. This process obfuscates a program by changing the structure of only a small portion of the program per iteration, but many iterations produce a randomized program.

For this nascent research, we narrow our focus to combinational Boolean circuits. This simplifies the problem domain by avoiding non-terminating programs and program state (memory). Additionally, circuits can be modeled using constructs from the mathematical discipline of graph theory.

1.2 Research objectives

We seek to accomplish two objectives with this research.

1. Develop a software architecture for developing and testing random selection schema for obfuscating a circuit's structure.
2. Develop an initial set of selection algorithms and characterize their behavior with regards to white-box obfuscation.

The first objective above is a means to an end. In other words, to develop and analyze selection algorithms, we need an architecture which will import, export, and manipulate combinational Boolean circuits. No complete application is available to perform the operations we seek to employ, so we developed a software package (CORGI[1]) to fill the void. Although CORGI is all new, it integrates an existing Java library (JGraphT) to represent the circuits as directed acyclic graphs.

For the second objective, we devised candidate algorithms which demonstrate the concept of random selection and replacement. The algorithms each produce an obfuscated version of an original circuit. Each circuit produced in this way is a randomly "selected," semantically equivalent version of the original, with the selection occurring as a sequence of steps rather than a single-step selection from a large set.

Although this research is based on a new obfuscation paradigm, the next chapter explores the current theoretical understanding of obfuscation and how it relates to our current work.

---

[1] CORGI stands for *C*ircuit *O*bfuscation via *R*andomization of *G*raphs *I*teratively, and is discussed in more detail in Section 3.3.1

II. Literature Review

Several key papers have been published which provide theoretical bases for why obfuscation is both impossible and, indeed, possible. Practical applications of these theories, however, do not appear in the literature. As such, one approach, the Random Program Security Model, proposes that practical obfuscation is indeed possible and that a program's intent can be protected even if the adversary has access to the obfuscated version of the program. The Random Program Security Model is fundamentally an analog to data encryption, but applied to programs rather than data.

2.1 What is obfuscation?

2.1.1 Preliminary definitions. Before delving into the finer details of obfuscation, it is instructive to understand how the word *obfuscation* is used in several contexts. In generic speech, to obfuscate means to "make obscure" or "confuse" [13].

As applies to computing, to obfuscate means "to alter code while preserving its behavior but conceal its structure and intent" [19]. Alternately, obfuscation is "any efficient semantic-preserving transformation of computer programs aimed at bringing a program into such a form, which impedes the understanding of its algorithm and data structures or prevents the extracting of some valuable information from the plaintext of a program" [18]. These two definitions provide the context for our review of current theory and techniques for program obfuscation.

2.1.2 Classifications of obfuscation. Program development and execution involves several steps, and program obfuscation can be applied at one or more of these steps. Fundamentally, there are three classifications of program obfuscation: layout, data, and control [3]. *Layout* obfuscation involves such techniques as scrambling identifier names and removing layout formatting. Both of these techniques operate on the source code, and do nothing to alter control flow of the program.

Data obfuscation is also primarily focused on altering the source code. Techniques include (a) storage and encoding transformations, which alter the way data is encoded or manipulated (b) aggregation transformations, which operate on data structures, and (c) ordering transformations, which change the order of variables and methods (within classes) and parameters (within methods). To some extent, these techniques can have an impact on control flow within a program, but it is not the primary intent. Like layout obfuscation, many of the specific transformations do not change control flow (although some introduce new control mechanisms).

The final classification is *control* obfuscation, and its techniques include (a) control *aggregation* transformations, which break up computations that logically belong together or merge computations that do not, (b) control *ordering* transformations, which randomize the order in which computations are carried out, and (c) control *computation* transformations, which insert new (redundant or dead) code, or make algorithmic changes to the source application. Control obfuscation techniques, as described in [3], are not strictly limited to source code, which means it has more generic applicability (e.g., assembly language and machine code).

Among the three broad categories described above, general program (circuit) obfuscation must account for control flow. This becomes clear as we look at additional definitions of obfuscation.

2.1.3 Theoretical definitions. The first formalized theoretical definition of program (or circuit) obfuscation was introduced by Barak et al. in [1]. This was a watershed publication because it formally proved that universal obfuscators do not exist. It also had the effect of spawning alternate theoretically-based definitions of obfuscation in several publications which followed. We will look at several of these definitions here.

2.1.3.1 Virtual Black Box Obfuscation. "Informally, an obfuscator $\mathcal{O}$ is an (efficient, probabilistic) *compiler* that takes as input a program P (or circuit C)[1] and produces a new program $\mathcal{O}(P)$ that has the same functionality as P yet is *unintelligible* in some sense" [1]. In lay terms, virtual black box (VBB) obfuscation can be thought of as some transformation to a program which completely hides all information about the program except input/output (i.e., black box) behavior, even though the obfuscated program is itself observable. In that sense, the obfuscated version provides virtually equivalent information as could be obtained with only black box access to the program.

Although informal, the definition above makes no distinction of what constitutes a program. No mention is made of "source code," "assembly language," or "machine code" anywhere in the paper (save one quote in a footnote). Thus, while there are clear differences between the three levels of a program, their fundamental nature is the same. Indeed, their equivalence is evidenced by the fact that programs can be viewed as boolean (specifically, combinational) logic circuits, and the Barak paper uses the terms *program* and *circuit* almost interchangeably. This is not to imply that obfuscated source code will necessarily yield object code that is obfuscated to the same degree (however measured). This remains an open question which, in part, will be addressed by this thesis.

Barak et al. formally define a (circuit) obfuscator as having these three properties:

1. *Functionality* property: For every circuit C, $\mathcal{O}(C)$ describes a circuit that computes the same function as C.
2. *Polynomial slowdown* property: There is a polynomial p such that for every circuit C, $|\mathcal{O}(C)| \leq p(|C|)$. This property may apply to size, run time, or both.

---

[1] Since this concept applies equally to programs and circuits, and since this thesis will specifically explore obfuscation of circuits, we will limit further discussion to circuit obfuscation. Therefore, substituting C for P does not alter the definition.

3. “*Virtual black box (VBB)*” property: For any probabilistic polynomialtime Turing machine (PPT) A, there is a PPT S and a negligible function α such that for all circuits C,

$$|\Pr[A(\mathcal{O}(C)) = 1] - \Pr[S^C(1^{|C|}) = 1]| \leq \alpha(|C|) \tag{2.1}$$

The obfuscator $\mathcal{O}$ is efficient if it runs in polynomial time.

From this definition, Barak, et al. prove that no universal obfuscator exists. The basis of their proof is to show that, for any given obfuscator, there exists a family of circuits which cannot be obfuscated. “However, it does not mean that there is no method of making circuits ‘unintelligible’ in some meaningful and precise sense” [1]. To be clear, the impossibility result still allows for a given obfuscator $\mathcal{O}$ to be able to protect some (though not all) families of circuits $\mathcal{C}$. From this, Barak et al. offer a weaker notion of obfuscation: indistinguishability obfuscation.

2.1.3.2 Indistinguishability Obfuscation. An indistinguishability obfuscator is defined in the same way as a circuit obfuscator, except that the “*virtual black box*” property is replaced with the following:

- *Indistinguishability* property: For any PPT A, there is a negligible function α such that for any two circuits C_1, C_2 which compute the same function and are of the same size k,

$$|\Pr[A(\mathcal{O}(C_1))] - \Pr[A(\mathcal{O}(C_2))]| \leq \alpha(k) \tag{2.2}$$

Observe that the indistinguishability property compares the obfuscations of two different circuits, unlike the VBB property, which compares an obfuscated circuit to a simulator which has only black box access to the original circuit. By weakening the VBB definition in this way, it is provable that obfuscation (however inefficient) is not impossible.

2.1.3.3 Best-Possible Obfuscation.

Goldwasser and Rothblum define an obfuscator as "a compiler that transforms any program (which we will view... as a boolean circuit) into an obfuscated program (also a circuit) that has the same input-output functionality as the original program, but is unintelligible" [6]. It is clear that this is the same definition found in [1], but it is nevertheless included because of the parenthetical comment that programs can be viewed as circuits. This concept is central to the research presented herein.

2.1.4 Practical applications.

Obfuscation software, of varying sophistication, is widely available from both commercial vendors and open source developers. Among commercial products, there are several well-known titles. PreEmptive Solutions [16] produces two popular tools: Dotfuscator (for .NET) and DashO (for Java). Smardec [17], produces Allatori, a Java obfuscator. Yet another company, Semantic Designs, Inc. [15] has a suite of tools collectively called Thicket™. It provides tools to obfuscate several languages, including C, C++, C#, Java, JavaScript, Ada, and PHP. There are, of course, other vendors which offer products that purport to obfuscate software to some degree, but enumerating them all here is beyond the scope of this thesis.

On the open source side, the number of projects is as plentiful as on the commercial side. One in particular, ProGuard Java Optimizer and Obfuscator is one of the most popular projects on SourceForge.net.[2]

It is not surprising that these companies and open source developers reveal little about the inner workings of their obfuscation techniques, except to describe the results of applying a particular approach (e.g., name obfuscation, flow obfuscation, string encryption, etc.). Interestingly, however, Semantic Designs' web site unequivocally states, "Warning: obfuscators do not stop reverse-engineering efforts by really determined opponents." This statement is an acknowledgment of the theoretical work

---

[2] From its home page, "SourceForge.net is the world's largest Open Source software development web site." As of 16 Jan 2008, ProGuard was ranked 291 out of 166,996 projects listed.

of Barak et al. described above. Nevertheless, practical obfuscators are not in short supply, despite this limitation, which begs the question: "Why not?"

2.2 Shortfalls of current theoretical work

To begin to answer the question of why practical software obfuscators are even available, much less *trusted*, one must further ask, "what makes them useful despite the impossibility results asserted—'*proved*'—by the theoreticians?" The answer is at least two-fold.

First, commercial and open source obfuscation tools are not typically employed, for the most part, to hide the purpose of the target software, but rather to hide the manner in which that purpose is achieved. For example, Microsoft may choose to obfuscate all or part of the source code for its spreadsheet program, Excel™. The obfuscated version would not hide the fact that the application is a spreadsheet. Rather, it would hide some portion of the code to prevent competitors from learning how part of the code is implemented, thus protecting Microsoft's competitive advantage in the marketplace. In this way, the obfuscation would be useful, even if though it necessarily fails the VBB paradigm of perfectly secure obfuscation.

A second (perhaps more profound) reason may be that the tools do not address obfuscation from a theoretical perspective. In light of an absence in the literature that correlates theoretical results to practical implementations, it is difficult to make this claim definitively (i.e., "absence of proof is not proof of absence"). It is nonetheless intriguing that developers do not relate the strength of their obfuscation schema to results predicted by the theoretical models.

From a VBB perspective, no obfuscators of any ilk should be useful or beneficial. Although the VBB standard is not achievable in a general, efficient, universal sense, some amount of obfuscation, as pertains to some *as-yet undefined* metric of obfuscation, may be desirable. This is certainly the case with existing obfuscators, even if not explicitly stated or understood by the developers, because all such tools both *exist* and *fail the VBB test*. Therefore, the VBB standard is not viable as a measure of practical obfuscation.

The other two theoretical results mentioned before—*indistinguishability* obfuscation and *best-possible* obfuscation—are similar. They both relate obfuscation to some *property* of the program, and use that to compare obfuscation results to each other (whereas VBB relates obfuscation to a black box version of a program). This distinction is subtle, but it opens the door to finding useful obfuscators even if they fail VBB scrutiny. Unfortunately, the underpinning theory behind indistinguishability obfuscation and best-possible obfuscation do not offer suggestions on what property or properties of a program should be the basis of comparison when deciding if an obfuscator yields indistinguishable results, or the best-possible level of obfuscation.

The research supporting this thesis was conducted to directly address what properties of a program might (or might not) be useful measures of obfuscation, and to provide a framework for empirically testing the efficacy of those properties. In other words, we seek to produce a "tangible" correlation to the theoretical work which has preceded this research. This objective is an outgrowth of the doctorate research conducted by Lt Col Todd McDonald. In his dissertation, he suggests a new paradigm of program obfuscation, the Random Program Security Model [12].

2.3 Random Program Security Model

Recall from [1] the theoretical benchmark definition of an obfuscator—the VBB paradigm—requires that three properties hold: functionality, polynomial slowdown, and the VBB property. Under the Random Program Security Model (or simply Random Program Model, RPM), McDonald replaces two of the three properties, functionality and VBB [12]. Only the polynomial slowdown property is retained.

For the functionality property, McDonald postulates instead that program obfuscation should apply both black-box and white-box obfuscation techniques. The principle is that neither approach on its own is sufficient to obfuscate a program.

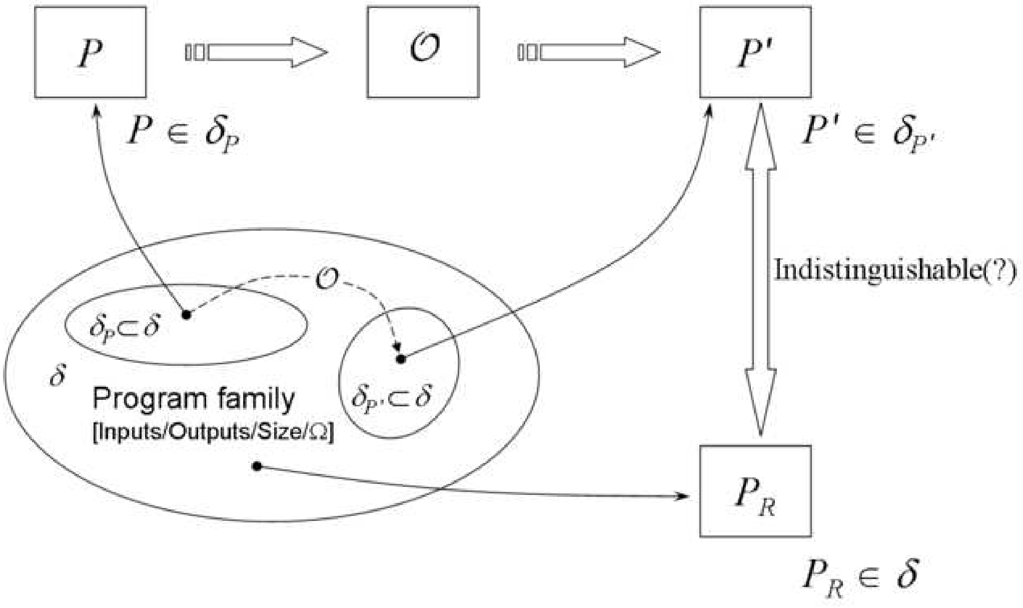


Figure 2.1: The Random Program Model (Program domain)

When combined, however, they act synergistically to overcome the inherent weaknesses of each.

For the VBB property, McDonald reasons that if an obfuscated program is indistinguishable from another program randomly-selected from the same family of programs (based on inputs, outputs, and size of the program), then the intent of the original program is protected.

The RPM is similar to, and derived from, data cryptography. RPM models black-box obfuscation after data encryption, and white-box obfuscation is analogous to comparing cryptographic data ciphers to random bit strings. Figure 2.1 graphically depicts the RPM. The obfuscator function, $\mathcal{O}$, uses both black-box and white-box transforms, as shown in Figure 2.2. These are described below in Sections 2.3.1 and 2.3.2.

2.3.1 Program encryption. Figure 2.3 illustrates the concept of black box obfuscation using program encryption. For an input x to program P, the result, $P(x)$ is the unobfuscated output of P. Intermediate result $P(x)$ becomes the input

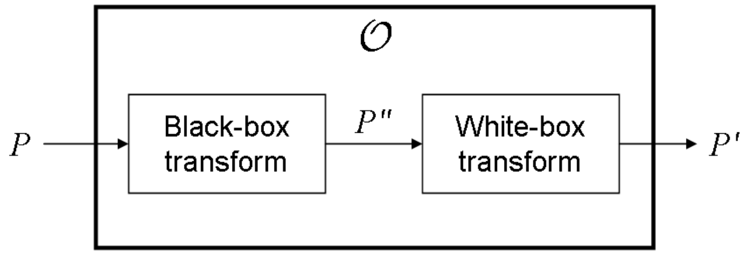


Figure 2.2: RPM obfuscation combines both black-box and white-box transforms

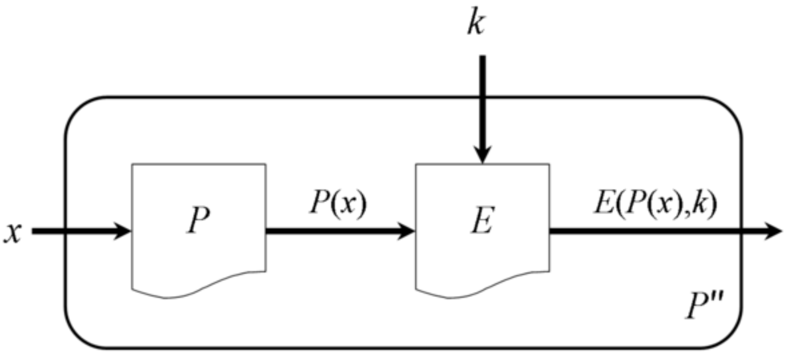


Figure 2.3: A black box obfuscation P'' of program P. P and P'' are not semantically equivalent because P'' includes a program, E, which encrypts the output of P.

of another component, E, which encrypts $P(x)$ based on some key k. The output $E(P(x), k)$ of E is the overall output of P''. Since $P(x) \neq E(P(x), k)$ (i.e., $P(x) \neq P''(x)$) for a given input x, program P'' is thus said to be a black-box obfuscated version of P.

Program encryption might be sufficient to protect a program if an adversary never obtains white-box access to the obfuscated program, P''. If the adversary *did* have white-box access, the demarcation between P and E would be discernible, and P would be revealed independent of E. Thus, RPM adds white-box protection to program encryption to achieve overall protection of the program's intent.

2.3.2 Intent protection. As previously stated, perfect, efficient, universal VBB obfuscators do not exist. If an adversary has access to an obfuscated, semantically equivalent program, the adversary will eventually be able to understand the intent of the original program. McDonald theorizes that program encryption can be augmented in such a way as to prevent an adversary from being able to isolate P from E in an encrypted program P''. The goal is to hide the fact that there is a semantics-altering component E. If this is possible, then even if the adversary is able to (eventually) predict the output of P'', such output will be meaningless with respect to $P(x)$, and program intent will remain protected.

McDonald proposes that if P'' (which is *not* semantically equivalent to P) is replaced with a randomly chosen—or produced—program P' (which *is* semantically equivalent to P''), then P is intent protected if the following hold:

- P' is such that the adversary cannot distinguish between the functional program P and the composite encryption program E
- P' is indistinguishable from a random program selected from the set of all programs the same size as P'

III. Methodology

The Random Program Model posits that an intent-protected program is indistinguishable from any other program with the same number of inputs and outputs, and of comparable size. This thesis specifically considers the white-box obfuscation component of the RPM. In this initial research, a program is modeled as a combinational boolean circuit. The circuit is white-box obfuscated by iteratively replacing random subcircuits with randomly-chosen, semantically-equivalent replacement subcircuits. Several algorithms are considered for selecting the subcircuits, and as well as candidate metrics with which to quantify the level of obfuscation achieved.

3.1 Notation

Since this research follows earlier work conducted by Lt Col Todd McDonald, we use his notation for the sake of consistency. Table 3.1 provides the notation used in the discussion which follows.

3.2 Assumptions

The current experimental environment relies on some simplifying assumptions, which are discussed here.

3.2.1 Programs represented as circuits. Software functionality, at its most fundamental level, can be represented as a sequence of Boolean expressions. For typical programs, which include loops (for, while, etc.), sequential boolean circuits map most directly to the program structure. In general, sequential (*cyclic*, in graph theory parlance) circuits can be converted to combinational (*acyclic*) circuits. Edwards [4] offers an algorithm which performs this transformation, but warns it is inefficient for anything but trivially small circuits (his algorithm ran for 51 seconds when operating on a 281-gate circuit). Despite potential intractability when converting large sequential circuits, we choose combinational logic over sequential logic because of its comparative simplicity.

Table 3.1: Notation used in describing the Random Program Model

Variable	Meaning
C	A combinational Boolean circuit
C'_i	Original circuit C after i iterations of randomization
C', C'_n	Original circuit C after n-iteration randomization is finished
Ω	circuit *basis*. Ω is a set of Boolean functions such that $\Omega \subseteq \{$AND, NAND, OR, NOR, XOR, XNOR, NOT$\}$
$C_{X\text{-}Y\text{-}S\text{-}\Omega}$	the *class* of a circuit, indicating inputs (X), outputs (Y), size ($S =$ maximum number of gates), and basis (Ω)
δ, $\delta_{X\text{-}Y\text{-}S\text{-}\Omega}$	circuit *family*, i.e., the set containing all circuits $C_{X\text{-}Y\text{-}S\text{-}\Omega}$
δ_C	family of circuits semantically equivalent to C ($\delta_C \subset \delta$)

The Random Program Model applies not only to the program domain, but to the circuit domain as well. Figure 2.1 is given again (with only a notational change) in Figure 3.1 to show the parallel between the two.

3.2.1.1 Combinational circuits. Combinational circuits have no state, whereas sequential circuits are temporal, which is to say they have memory and feedback loops (cycles). Since sequential circuits can be decomposed into combinational components, it is sufficient at the outset of this research to forgo the former in favor of the latter. As an aside, combinational circuits sidestep the issue of non-terminating programs–another complication of sequential circuits.

Our decision to use combinational circuits is supported by [9] which points out in Chapter IV that a very simple grammar is all that is needed to compute everything that can be computed by large languages like *C* and *Java*. In particular, the grammar, in Backus Naur form, is shown in Equation 3.1 where B represents any Boolean expression and E represents any integer expression. It is because of this

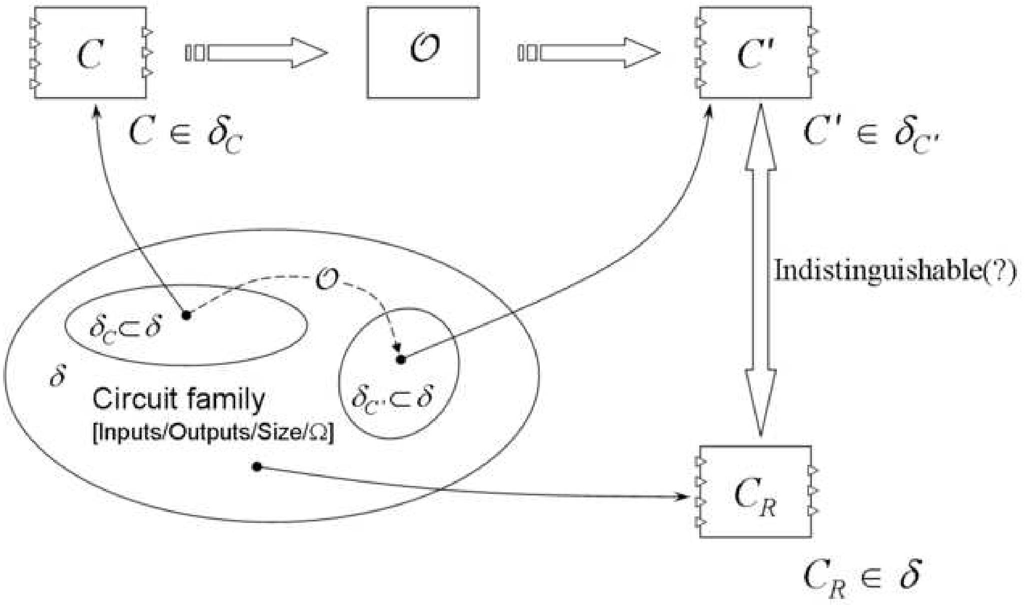


Figure 3.1: The Random Program Model (Circuit domain)

underlying simplicity that any software can be mapped to combinational logic form.

$$B ::= true|false|(!B)|(B\&B)|(B \parallel B)|(E < E) \tag{3.1}$$

An obvious benefit of choosing combinational logic is that it is easy to understand. As demonstrated in Equation 3.1 above, only three logic functions are necessary: NOT (!), AND (&), and OR (‖). There are other commonly used logic functions (namely NAND, NOR, XOR, and XNOR), but these can be represented using various combinations of NOT, AND, and OR.

Combinational logic circuits are used across a broad spectrum of applications, within both the hardware and software domains. At the 1985 International Symposium of Circuits and Systems (ISCAS), the IEEE introduced a set of benchmark circuits, which are collectively referred to as ISCAS-85 benchmark circuits. [8] They are particularly useful to our purpose, even though they were initially targeted at the hardware community. A list of these circuits can be found at [2]. The smallest of these circuits, C17, is shown in Figure 3.2.

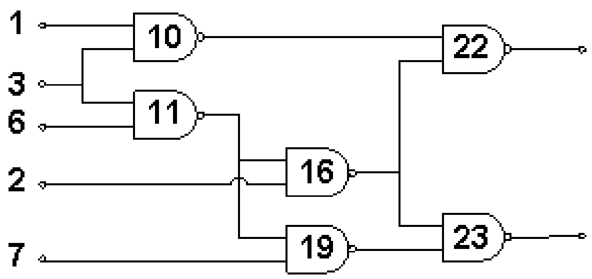


Figure 3.2: ISCAS Benchmark Circuit C17

3.2.1.2 Directed acyclic multi-graphs. In order to manipulate circuits, they must be in a format suitable for that purpose. For this research, the discipline of graph theory provides a suitable application domain. Namely, we represent circuits as ***directed acyclic multi-graphs***. We turn to Gross and Yellen [7] for a brief refresher on graph theory terminology to help describe the rationale for choosing graphs to represent circuits (reference Figure 3.3).

graph: A *graph* $G = (V, E)$ is a mathematical structure consisting of two finite sets V and E. The elements of V are called *vertices* (or *nodes*), and the elements of E are called *edges*. Each edge has a set of one or two vertices associated to it, which are called *endpoints*. [*Example*: All graphs in Figure 3.3.]

The authors correctly allow for edges with only one endpoint, which “is an edge that joins a single endpoint to itself.” However, such a construct in a circuit would make it sequential, not combinational. For our purposes, we only consider edges with exactly two distinct vertices. See the definition for *cycle* below.

directed edge: A *directed edge* is an edge, one of whose endpoints is designated as the tail, and whose other endpoint is designated as the head. An edge is said to be directed from its tail to its head.

directed graph: A *directed graph* (or ***digraph***) is a graph each of whose edges is directed. [*Example*: Figures 3.3(b), (d), and (f).]

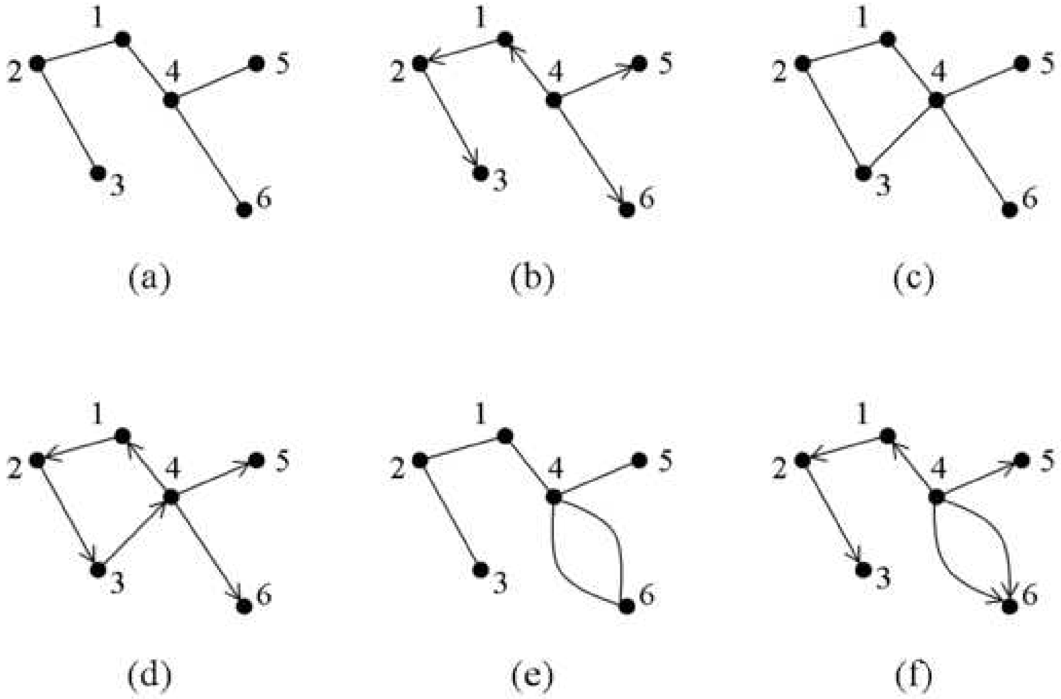


Figure 3.3: Example graphs.
(a) An *un*directed graph with no cycles.
(b) A *directed* graph with no cycles.
(c) An undirected graph with one cycle ($1-2-3-4-1$ *and* $1-4-3-2-1$).
(d) A directed graph with one cycle ($1 \to 2 \to 3 \to 4 \to 1$ *only*).
(e) An undirected multi-graph with one cycle.
(f) A directed acyclic multi-graph.

We must limit the graphs we use to directed graphs because in a combinational circuit, a connection between gates is always from the output of one gate to an input of another gate.

cycle: A *cycle* is a nontrivial closed path.[1]

acyclic graph: An *acyclic graph* is a graph that has no cycles. [*Example*: Figures 3.3(a), (b), and (f).]

Combinational circuits do not have any feedback loops or memory, as do sequential circuits. Therefore, only an acyclic graph can represent a combinational circuit.

multi-edge: A *multi-edge* is a collection of two or more edges having identical endpoints. The ***edge multiplicity*** is the number of edges within the multi-edge.

multi-graph: A *multi-graph* is a graph that may contain multi-edges. [*Example*: Figures 3.3(e) and (f).]

In a combinational circuit, it is permissible for the output of one gate to be connected to more than one input of another single gate. The analogous construct in graph theory is a multi-graph.

directed acyclic graph: A *directed acyclic graph* (DAG) is a graph that is at the same time a directed graph and an acyclic graph. It may or may not be a multi-graph. [*Example*: Figures 3.3(b) and (f).]

For our purposes, we implicitly accept DAGs as also being multi-graphs. In other words, *DAG* and *directed acyclic multi-graph* carry the same meaning, thus Figures 3.3(b) and 3.3(f) are both DAGs.

---

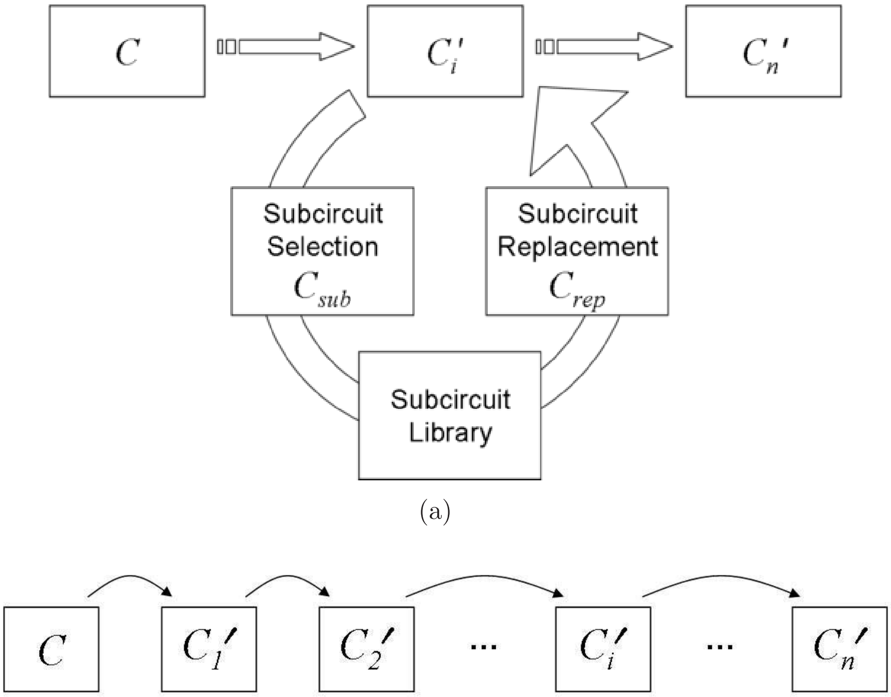
[1] A *path* does not repeat any vertex (except possibly the initial/final vertex) or edge. *Nontrivial* means the path includes more than one vertex. *Closed* means the initial vertex is the same as the final vertex.

3.2.2 Iterative randomization. The RPM requires that an intent-protected circuit, C', be indistinguishable from a randomly selected circuit, C_R. An interesting aspect of the RPM is that the comparison itself is not influenced by the choice of original circuit, C. Consequently, if the obfuscator $\mathcal{O}$ does *not* encrypt (i.e., semantically transform) a circuit, the indistinguishability comparison can still be performed. This fact allows us to segregate the white-box component of $\mathcal{O}$ from its black-box component as we explore randomization methods for white-box obfuscation of circuits.

To perform white-box obfuscation, we consider the process of subcircuit *selection* and *replacement*. Two reasons drive us to this choice. First, to randomly select a white-box replacement of C would require enumeration of *all* circuits in δ_C. As circuit size increases, δ_C becomes prohibitively large, and the obfuscator suffers greater-than-polynomial slowdown. Second, the separate steps of subcircuit selection and subcircuit replacement offer opportunities to inject randomness into the white-box obfuscation process.

Section 3.4.3 describes selection and replacement in greater detail, but we introduce here the basic of the concept (reference Figure 3.4). Given a circuit C which is to be white-box obfuscated, select a subcircuit, C_{sub}. Retrieve a randomly chosen circuit C_{rep} from a library of circuits which contains a set of *all* circuits semantically equivalent to C_{sub} (the assumption that such a library exists will be discussed in Section 3.2.3). Finally, remove C_{sub} from C and insert C_{rep} in its place. As long as C_{sub} and C_{rep} are semantically equivalent (and the order of inputs and outputs is preserved), then semantic equivalence exists for C, all C'_i, and C'_n.

3.2.3 Circuit library exists. A library of replacement circuits must exist in order for the process of iterative randomization to be possible. However, in Section 3.2.2 we said that enumerating all possible replacements for C would violate the polynomial slowdown condition of RPM. We overcome this apparent contradiction



(a)

(b)

Figure 3.4: Two representations of iterative white-box randomization.
(a) White-box obfuscation of circuit C by iteratively replacing randomly selected subcircuits (C_{sub}) with a semantically equivalent subcircuit (C_{rep}) chosen randomly from a circuit library. C is the unobfuscated circuit, C'_i is C after the ith iteration of replacement, and C'_n is C after an n-iteration obfuscation is complete.
(b) Depicts the sequential iterations of subcircuit selection and replacement.

by developing[2] a library whose contents are limited to only small circuits, typically on the order of 5 or fewer gates. In this way, all semantically equivalent circuits in a particular family (i.e., all $C \in \delta_C$) *can* be enumerated. Therefore, in the iterative replacement process, a given C_{rep} can truly be selected from among all size-bounded circuits semantically equivalent to C_{sub}.

3.3 Obfuscation toolkit

As this research is empirically based, a software tool was developed to perform the white-box circuit obfuscation portion of the RPM. Although the RPM calls for both black-box (program encryption) and white-box (randomization) techniques, they are performed independently from one another. This allows us to develop software which only performs the white-box function. The tool has two major components, CORGI and CXL.

3.3.1 CORGI: the circuit randomizer.

CORGI, which stands for *C*ircuit *O*bfuscation via *R*andomization of *G*raphs *I*teratively, was developed to empirically analyze the RPM. Its development was a major benefit of this research. The inner workings of the software are described in greater detail in Appendix A. Here, we briefly discuss the main features of CORGI.

3.3.1.1 Development environment.

CORGI is coded entirely in Java. Several factors influenced this choice. First, there is a strong emphasis on object-oriented design (OOD) at the Air Force Institute of Technology (AFIT), and Java is the de facto language of choice for the academic environment. Second, given the nature of the problem domain (i.e., circuits), OOD is a logical design choice. The third factor is based on our choice of application domain (i.e., to represent circuits as graphs), which allowed us to incorporate JGraphT into the development.

---

[2] The circuit library used in this research is a product of concurrent research conducted by Capt Moses James. His research focuses on circuit randomization as a set selection problem.

Table 3.2: The most notable features and benefits JGraphT contributed to the development of CORGI.

Feature	Benefit to CORGI development
Graph package	Model CORGI circuits as graphs. In particular, JGraphT's graph package included classes for all the types of graphs described in Section 3.2.1.2.
Subgraph class	Manipulate subgraphs without modifying the base graph. This is a critical component of the subcircuit selection and replacement process.
Exporter classes	Export circuits to standard formats used by various graph software packages (e.g., yGraph, GraphVis, prefuse, etc.). Allows user to render circuits visually.
Algorithms package	Contains classes for standard algorithms used in graph theory. In particular, the CycleDetector class is a critical part of CORGI because it enforces the acyclic nature of DAGs.

JGraphT is an open source Java graph library [14]. Its free availability as an open source project shortened the time to develop CORGI by at least several weeks–possibly much more. JGraphT provides the means to easily generate graphs and apply to them many of the common graph theory techniques. It is the crux of what makes CORGI work. JGraphT not only provides the ability to model the underlying graph of a circuit, it also has methods and services which make circuit manipulation and analysis possible. Table 3.2 shows the key features and benefits of JGraphT.

Despite the graph basis for circuit manipulation—as implemented by way of the JGraphT library—CORGI completely elides from the user any references to graphs or graph behavior. Thus, CORGI is effectively a translation between the two domains.

3.3.1.2 Subcircuit selection and replacement. Subcircuit selection and replacement is the principle function CORGI performs. From the user perspective, it is a single action, but as already described, this function is iterative. We describe in more detail here the mechanics of how CORGI carries out one iteration of the process (ref. Figure 3.1).

CORGI does not actually select subcircuits. Instead, it selects a subset of the circuit's gates based on a selection strategy chosen by the user.[3] This subset of the circuit's gates corresponds to a subset of vertices in the underlying graph, by which a *vertex-induced subgraph* (or simply *subgraph*) is derived. CORGI then copies the subgraph (leaving the base graph unchanged) and uses it to construct a separate subcircuit representative of the gates selected.

Next, CORGI uses the new subcircuit's truth table, along with other user inputs, to request a replacement from the circuit library (CXL). CXL selects a random, semantically equivalent, subcircuit replacement (i.e., its truth table is the same). The original subcircuit is removed from the circuit, and the replacement subcircuit is inserted in its place.

3.3.2 CXL: the circuit library. CXL is a component of CORGI which contains a library of circuits. In a sense, CXL is really a library of *sets* of circuits. Each set is a circuit *family* δ_C where C is characterized by a particular *class* $C_{X\text{-}Y\text{-}S\text{-}\Omega}$ (ref. Table 3.1).

Because of the various equivalence relationships in Boolean logic, $|\delta_C|$ rapidly increases exponentially with even small increases in S and/or $|\Omega|$. For practical reasons, we choose $S \leq 3$, although we do allow $\Omega \subseteq \{\text{AND, NAND, OR, NOR, XOR, XNOR, NOT}\}$ (i.e., $|\Omega| \leq 7$).

From a user perspective, CXL is not a separate component from CORGI. Indeed, CXL is accessed by CORGI via an interface, which is called from within the iterative function of subcircuit selection and replacement. The user provides parameters which are used by the interface, but the call itself is not controlled by the user. Because of this, we consider CXL to be an integrated component of CORGI, and this perspective is implicit in any further references to CORGI unless otherwise stated.

See [10] for more detailed information on the behavior of CXL.

---

[3] The initial implementation of CORGI limits selection to only one or two gates, primarily for performance reasons, but also due to limitations imposed by the circuit library.

3.4 Empirical Approach

This research is predicated on the notion that we need empirical data to be able to demonstrate whether practical obfuscation might be possible in light of theoretic impossibility results. Perhaps there exist imperfect obfuscators that protect circuits to a useful, measurable degree. Inherent in the preceding conjecture are two questions:

- What properties of circuits are indicators of *useful*, measurable circuit protection?
- What methods of obfuscation produce such properties in circuits?

Since our standard of *useful* is the RPM, we are really asking what properties of circuits are indistinguishable between an obfuscated circuit, C', and a randomly selected (generated) circuit, C_R. If we know which properties relate to indistinguishability under the RPM, our intuition is we should be able to easily find algorithms which produce those properties in C'. On the other hand, if we know that a particular obfuscator will produce a C' which meets the RPM definition of indistinguishability, we can deduce which properties are indicators of well-obfuscated circuits.

In reality, we do not know *a priori* the answer to either of the two questions above. Our approach, therefore, is to work the problem incrementally to see where the results converge. We briefly consider several candidate properties with which to measure circuit obfuscation under RPM. Then we propose several algorithms for performing subcircuit selection as part of the iterative randomization process. These algorithms are applied to a circuit, C, and then the resulting white-box obfuscated circuit, C', is examined for their effect on obfuscation under RPM. Next, we define some key concepts used in the discussion which follows.

3.4.1 Key concepts. First, a circuit *property*, as we shall use the term, is a descriptor of a single circuit. This is an important distinction since the white-box circuit obfuscation process we employ is iterative (ref. Figure 3.4), creating many intermediate circuits ${C_i}'$ before finishing with ${C_n}'$ (${C_n}'$ is the same as C' in

Figure 3.1). These intermediate circuits provide us the means to measure how a given property changes throughout the iterative process, but each C_i' will have its own set of properties independent of any other circuit.

Second, since combinational circuits are modeled as DAGs, we look initially to graph theory for properties of *graphs* which may be candidate measures of *circuit* obfuscation. This choice leads us to also use graph terminology to describe some of the properties. When this occurs, equivalent terminology—if it exists—is included parenthetically.

Third, our use of the term *path* is limited to only those paths which begin at a circuit input and end at a circuit output. The intention is to describe control flow through a circuit.

Fourth, DAGs are by their nature hierarchical, thus combinational Boolean circuits are, too. A circuit's gate *hierarchy* is dictated by the predecessor or successor relationships of the various gates in the circuit. By our convention, if a gate precedes another gate in some path through the circuit, then the preceding gate is at a higher level. Equivalently, if a gate succeeds another gate along some path through the circuit, then the succeeding gate is at a lower level. It is possible that a particular gate could be assigned to any one of several levels, but our convention is to assign the gate to the lowest level that preserves the hierarchy of the circuit.

Figure 3.5 demonstrates the concept of gate hierarchy. Note that gate B is at level 2, not level 1, as is gate C. This is because the longest path from inputs of gate B to the output of gate D is length 2. Similarly, gate A could have been assigned to a new level 3, but the addition of the extra level breaks the convention that gates should be assigned to the lowest level that preserves the hierarchy of the circuit.

Finally, certain proposed circuit properties are frequency distributions, represented graphically as histograms. An example might be the number of unique paths that transit each gate. In Figure 3.5, for example, gate A has two unique paths:

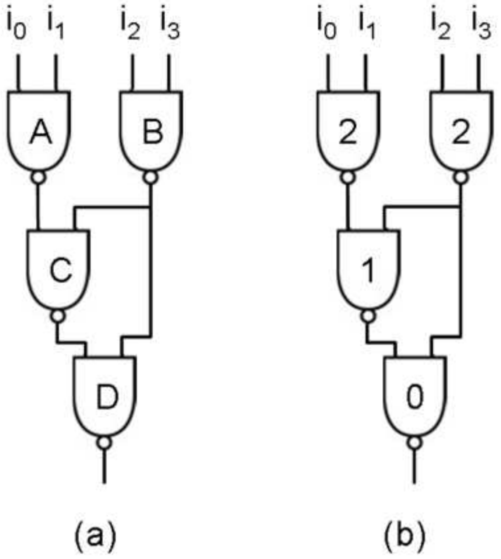


Figure 3.5: A simple example of circuit hierarchy.
(a) A simple circuit ($X = 4$, $Y = 1$, $S = 4$, $\Omega = \{\text{NAND}\}$) without hierarchical levels.
(b) The same circuit with lowest hierarchy level assigned to each gate.

i_0-A-C-D and i_1-A-C-D. Similarly, gate B has two, gate C has four, and gate D has six. The associated histogram is shown in Figure 3.6.

3.4.2 Properties of obfuscated circuits. A property of a circuit may be a single value (e.g., average path length), or a *distribution* of values (see Figure 3.6). In case of the latter, the property will be identified as such. We propose several circuit properties as candidate measures of circuit obfuscation, without consideration of the efficacy of each property (see Table 3.3.)

To be clear, the properties listed in Table 3.3 serve two purposes. First, they are objects of the proposed algorithms (Section 3.4.3 below). Second, they are collectively a leaping-off point for future research on which circuit properties are strong indicators of effective obfuscation.

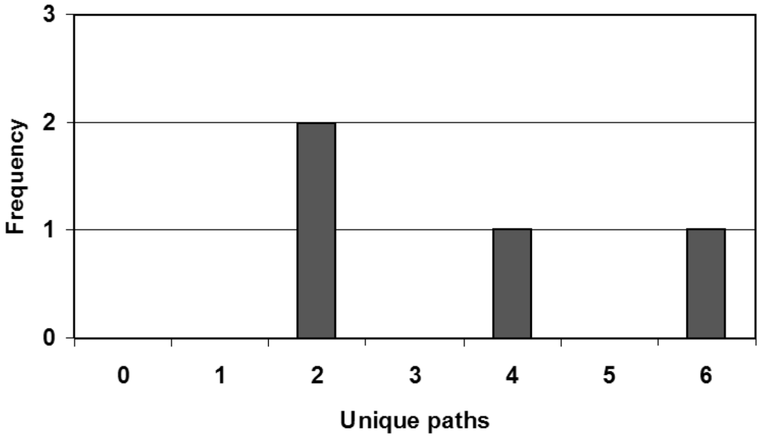


Figure 3.6: A simple example of a histogram of a circuit property. The chart represents the frequency of occurrence of gates having a particular number of unique paths passing through them. In this example circuit, two gates have two unique paths (gates A and B), one gate has four unique paths (gate C), and one gate has six unique paths (gate D).

Table 3.3: A set of candidate circuit properties for measuring circuit obfuscation.

Circuit-level	Gate-level
Number of vertices at each hierarchical level [distribution]	Number of paths through each gate [distribution]
Set of input/output pairs as determined by paths through the circuit	Number of unique input/output pairs represented by paths through each gate [distribution]
Number of vertex (gate) types ($\|\Omega\|$)	Number of successors of each gate (i.e., gate *fanout*) [distribution]
Number of each vertex type (e.g., AND, OR, etc.) [distribution]	Number of predecessors of each gate (i.e., gate *fan-in*) [distribution]

Table 3.4: A set of candidate subcircuit selection algorithms used to iteratively white-box obfuscate a circuit. Algorithm names are derived from the file name of the Java class which implements the algorithm in CORGI.

Selection Algorithm	Description
`RandomSingleGate`	Selects a single gate at random
`RandomTwoGates`	Selects two gates at random
`RandomLevelTwoGates`	Selects a hierarchical level at random, and limits replacement to two gates selected at random from that level (±1 level)
`FixedLevelTwoGates`	Same as `RandomLevelTwoGates` except the hierarchical level is specified
`LargestLevelTwoGates`	Same as `FixedLevelTwoGates` except the hierarchical level is the one containing the most gates
`OutputLevelTwoGates`	Same as `FixedLevelTwoGates` except the hierarchical level is 0 (level 0 contains all the output gates)

3.4.3 White-box obfuscation algorithms. CORGI was designed to use multiple, interchangeable subcircuit selection algorithms. Recall that under the RPM, an obfuscated circuit C', which is semantically equivalent to circuit C, is indistinguishable from a random circuit C_R. We would like to be able to select C' from a completely enumerated $\delta_{C'}$, but for large $|C|$, the size of $\delta_{C'}$ is prohibitively large to enumerate all circuits in the set. This limitation forces us to choose another method of random "selection" of C': iterative randomized subcircuit selection and replacement.

The process of obfuscating a large circuit by iteratively randomizing small subcircuits provides opportunities and introduces challenges as compared to direct selection from $\delta_{C'}$. An advantage of the process is that a subcircuit selection algorithm can be chosen such that it optimizes a particular obfuscation metric. A disadvantage, due to the fact that the process is a metaheuristic, may be that a particular sequence of iterations will converge on a final C' with a suboptimal value for the target property.

Table 3.4 lists a candidate set of randomization algorithms developed for this research with a brief description of each. In Chapter IV, we analyze these algorithms and how they were derived.

IV. Results

CORGI is an architecture for obfuscating combinational Boolean circuits via iterative subcircuit selection and replacement. Six strategies for *subcircuit selection* are implemented in CORGI as modular *algorithms.* When executed, these algorithms transform a circuit C into a randomized (i.e., white-box obfuscated) but semantically equal circuit C'. The nature of the transformation is different for each algorithm. Also, the design of certain CORGI components degrades CORGI performance (runtime) when some selection algorithms are employed.

4.1 Overview

To perform white-box obfuscation under the RPM on circuit C, we would ideally like to enumerate all circuits in δ_C, then select one at random as the semantically equivalent replacement circuit C'. Such enumeration is infeasible for large circuits, which means a replacement circuit cannot be directly selected at random. Instead, it must be *built*, but still yield a random $C' \in \delta_C$. The process of iterative subcircuit selection and replacement described in Section 3.2.2 provides two ways for introducing randomness into the process.

1. *Random selection*: Select a subcircuit $C_{sub} \subset C$ at random.
2. *Random replacement*: Select a replacement circuit $C_{rep} \in \delta_{C_{rep}}$ at random.

There may also be some intermediate circuit C'_i for which non-random selection and replacement are preferred. Here, also, there are two such *smart* choices.

1. *Smart selection*: Only select subcircuits which have a particular property. If the subset of allowable selections contains more than one subcircuit, then one may be selected at random or based on another property.
2. *Smart replacement*: Similar to smart selection, only select replacement circuits from the library which have a particular property. If the subset of allowable selections contains more than one subcircuit, then one may be selected at random or based on another property.

4.2 Limitations

Our research exposed certain limitations on the development of subcircuit selection and replacement algorithms. *Smart* strategies often impinged upon temporal or spatial efficiency, and the problem domain (i.e., combinational Boolean circuits) reduced the randomness of *random* selection strategies as we seek to avoid creating sequential circuits.

4.2.1 Smart strategies. There are multiple ways to make smart subcircuit *selections*. Some examples include choosing only subcircuits with a particular input size (X_{sub}), output size (Y_{sub}), circuit size (S_{sub}), basis (Ω_{sub}), and/or truth table. Selection can also be made based on particular subsets of the circuit's gates. For example, select only subcircuits which have gates in a particular hierarchical level in the circuit. Other smart selection strategies require searching the underlying graph for isomorphic subgraphs, which is an NP-complete problem [5]. These can all pose intractability problems for our iterative randomization process when we have large circuit sizes, which limits the efficiency of the search algorithm.

Consider a smart selection strategy which is based on subgraph isomorphism. Since the search is NP-complete, and the search space can be quite large (circuits with thousands, perhaps millions of gates), the strategy becomes too computationally intensive to be efficient, as required by the RPM.

Two of the six algorithms (`RandomSingleGate` and `RandomTwoGates`) use a purely random selection strategy which are discussed in Sections 4.3.2 and 4.3.3. The other four algorithms (`RandomLevelTwoGates`, `FixedLevelTwoGates`, `LargestLevelTwoGates`, and `OutputLevelTwoGates`) use a blend of smart and random selection, as is described in Sections 4.3.4–4.3.7. None of the latter four algorithms use NP-complete selection strategies.

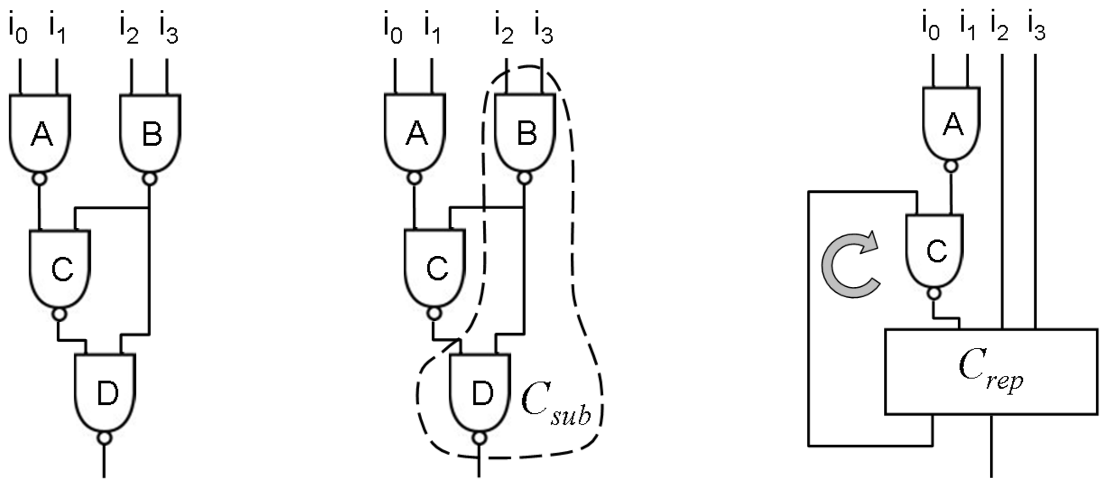
As for smart *replacement*, CXL currently has no means to employ such a strategy. The problem is more a limit on space than on time. Specifically, if all replacement circuits are stored with sufficient metadata, then finding a particular replacement is basically a simple lookup in a database. However, as the size bound of candidate replacement circuits increases, the size of the library increases exponentially, thus limiting the set from which replacements can be selected.

4.2.2 Introduced cycles. The choice of combinational Boolean circuits places a particular limitation on which subcircuits may be selected for replacement, as stated in Axiom 1.

Axiom 1. *In order to maintain the combinational structure of circuit C, the set of gates $G(C_{sub})$ in a selected subcircuit C_{sub} must not contain* any *pair of gates (G_i, G_j) such that (WLOG):*
(a) G_i precedes G_j along some directed path in C, and
(b) the longest *directed G_i-G_j path in C is ≥ 2.*

The results of improperly selecting C_{sub} is shown in Figure 4.1. Figure 4.1(a) shows a 4-input, 1-output, 4-gate circuit. In Figure 4.1(b), a 3-input, 2-output, 2-gate subcircuit C_{sub} is selected for replacement. However, C_{sub} contains a pair of gates, B and D, which violate Axiom 1. Figure 4.1(c) shows that a cycle is created if C_{sub} is replaced with any replacement circuit C_{rep}. The problem occurs because gate C receives an *out*put from C_{sub} but also provides an *in*put to C_{sub}, thus creating a cycle. If C_{sub} is improperly selected, there exists no C_{rep} such that a cycle is not created.

As a result of this limitation, the manner of subcircuit selection in CORGI requires a sequential selection of gates for those algorithms which select multi-gate subcircuits. There are differences in how this is performed for each algorithm, which are discussed below. The important point here is that, once the set of gates is selected, the subcircuit is defined (induced) by the set of selected gates, as well as all connections ("wires") leading into or out of those gates. It is not necessary that selected gates be connected directly to each other in C.



(a) (b) (c)

Figure 4.1: An example of an improper subcircuit selection and how it will create a cycle after replacement.
(a) A circuit before subcircuit selection.
(b) Subcircuit C_{sub} is selected. It is not a valid selection since gate B is a predecessor of gate D *and* the longest path from B to D is ≥ 2.
(c) A cycle is created after replacing an improperly selected subcircuit, regardless of what replacement C_{rep} is used.

4.3 Analysis of subcircuit selection algorithms

For this research, six subcircuit selection algorithms were developed. All algorithms adhere to a standard selection interface in CORGI, which does not actually select a subcircuit C_{sub} directly from a circuit C'_i. Instead, the interface requires each algorithm to return a set of gates G. CORGI then uses G, together with JGraphT's `DirectedSubgraph` class to create the subcircuit C_{sub} induced by the selected gates in set G. Thus, each algorithm returns a set of gates, not a subcircuit. The sections that follow describe the manner of selection and the "behavior" each algorithm exhibits.

The development of these algorithms was itself an iterative process. As each new algorithm was developed and tested, the results would suggest alternate strategies for selection. Therefore, the algorithms are presented below in roughly the same order in which they were developed.

4.3.1 Common functions. The overall process of randomization is also presented in Appendix A (Algorithm 7). For the sake of brevity here, we defer to Appendix A, Section A.2 for the details of two functions used by the six selection algorithms discussed below: `SelectRandomGate` (Algorithm 8) and `RejectGates` (Algorithm 9). It is sufficient to know that `SelectRandomGate` randomly selects a gate from a set of gates, and `RejectGates` populates a set of gates which should not be part of the input to `SelectRandomGate`.

A third function, `EstablishGateHierarchy`, is used only by the so-called *level* algorithms (those for which selection is based on a circuit's hierarchical level—all have "Level" in their name). The details of `EstablishGateHierarchy` are presented in Appendix A (Algorithm 10), but its basic functionality is to assign each gate to the lowest allowable level in the circuit's hierarchy. The details of why this function is required will be presented in Section 4.3.4, where we introduce the first of the level algorithms, `RandomLevelTwoGates`.

4.3.2 `RandomSingleGate`. `RandomSingleGate` was the first selection algorithm developed for CORGI. As the name implies, all subcircuit selections C_{sub} are of the class $C_{X_{sub}\text{-}1\text{-}1\text{-}\Omega_{sub}}$ where

$$\Omega_{sub} \subset \{\text{AND, NAND, OR, NOR, XOR, XNOR}\}$$
$$X_{sub} \geq 2$$
$$Y_{sub} = S_{sub} = |\Omega_{sub}| = 1$$

Since all C_{sub} contain only one gate, any gate can be selected from C'_i and replaced without creating cycles in C'_{i+1} (as long as C'_i is combinational). The selection procedure is simple, as shown in Algorithm 1.

`RandomSingleGate` was developed initially as a simple algorithm by which CORGI functionality could be tested. The function of removing a subcircuit from a circuit, then replacing it with a different subcircuit is a non-trivial activity. *Se-*

Algorithm 1 `RandomSingleGate`(C'_i)

```
$G_{sub} \leftarrow \varnothing$ {set of gates (1 in this case) to induce $C_{sub}$}
$G(C'_i) \leftarrow$ set of all gates in $C'_i$
$g_k \leftarrow$ call SelectRandomGate($G(C'_i)$)
$G_{sub} \leftarrow G_{sub} \cup \{g_k\}$
return $G_{sub}$
```

lecting a single-gate subcircuit for replacement, while simple to do, provides multiple dimensions by which to test the process of iterative randomization. When the subcircuit is replaced, gate properties such as type (NAND, NOR, etc.), fan-in (number of adjacent predecessors), fanout (number of adjacent successors), and whether the selected gate is a circuit output, must all be accounted for.

`RandomSingleGate` is a purely *random* (as opposed to *smart*) selection process. No knowledge of the target circuit is needed other than the set of gates in the circuit. The iterative process cannot be guided in any way.

There are three results produced by `RandomSingleGate`. First, no new external control flows are introduced in the circuit; second, the size S of C'_{i+1} is never smaller than C'_i; and third, the circuit becomes very "tall" ($\ell \propto n$, where ℓ = number of hierarchical levels, and n = number of replacement iterations).

The first result is contingent on how we use the term *control flow*. If we have access to the structure of a circuit, then every *unique* path through a circuit is a control flow. If, however, we only have black-box access to a circuit, then no distinction can be made between unique paths which share a common source (input) and sink (output). We will refer to the former as *internal control flow* and the latter as *external control flow*. `RandomSingleGate` will never introduce new *external* control flows because all subcircuit inputs connect to a single subcircuit output. However, `RandomSingleGate` will always introduce new *internal* control flows. The only way `RandomSingleGate` does *not* produce new internal control flows is the trivial case where the selected single-gate subcircuit is replaced with itself. All other semantically equivalent replacements have more than one gate, with connections between them; thus new internal control flows are always introduced.

The second result is a function of the replacement subcircuits C_{rep} returned by CXL. In order for a replacement C_{rep} of a single gate subcircuit C_{sub} to change C'_i, C_{rep} must have more than one gate. The reason for this is there is no non-trivial single-gate equivalence between any pair of gates (g_i, g_j) in $\Omega = \{$AND, NAND, OR, NOR, XOR, XNOR$\}$.

The third result is a natural consequence of the first two. A subcircuit C_{sub} comprised of a single Boolean logic gate has only one hierarchical level ($\ell = 1$). All nontrivial replacements C_{rep} of C_{sub} have at least two gates. If C_{rep} has n gates, then it can have $1 \leq \ell \leq n$ hierarchical levels. If $\ell \geq 2$, then C'_{i+1} could "grow"—relative to C'_i—by as much as $\ell - 1$ levels (although it may not grow at all). The rate of growth over many iterations is a function of which gates are selected and the average number of levels in each C_{rep} selected from CXL.

Reference Figure A.5(b) for a sample result of applying this algorithm to ISCAS benchmark circuit C17. As we see from the data presented in Section A.3, `RandomSingleGate` produces the tallest C' on average among all the circuits.

*4.3.3 **RandomTwoGates.*** `RandomTwoGates` is meant to be a two-gate version of `RandomSingleGate`. All subcircuit selections C_{sub} are of the class $C_{X_{sub}\text{-}Y_{sub}\text{-}2\text{-}\Omega_{sub}}$ where

$$\Omega_{sub} \subset \{\text{AND, NAND, OR, NOR, XOR, XNOR}\}$$

$$X_{sub} \geq 2$$

$$Y_{sub} = 1 \text{ or } Y_{sub} = 2$$

$$S_{sub} = 2$$

$$|\Omega_{sub}| = 1 \text{ or } |\Omega_{sub}| = 2$$

The selection of C_{sub} is accomplished by sequentially selecting the two gates. The first gate, g_1, is selected entirely randomly, in exactly the same fashion as the gate g_k was selected by the `RandomSingleGate` algorithm. The second gate, g_2, must be selected more carefully, however, in order not to introduce cycles after replacement. Specifically, g_2 can only be selected from a specific subset of gates in C'_i that remains after g_1 was selected (ref. Section 4.2.2). The procedure is shown in Algorithm 2.

Algorithm 2 `RandomTwoGates`(C'_i)

```
$G_{cand} \leftarrow \varnothing$ {set of candidate gates to select from randomly}
$G_{cand} \leftarrow G_{cand} \cup G(C'_i)$ {set of *all* gates in $C'_i$}
$g_1 \leftarrow$ **call** SelectRandomGate($G_{cand}$)
$G_{cand} \leftarrow G_{cand} - \{g_1\}$ {$g_1$ cannot also be $g_2$}
$G_{cand} \leftarrow G_{cand} - \{$**call** RejectGates($g_1$, **true**)$\}$ {remove predecessors of $g_1$}
$G_{cand} \leftarrow G_{cand} - \{$**call** RejectGates($g_1$, **false**)$\}$ {remove successors of $g_1$}
$g_2 \leftarrow$ **call** SelectRandomGate($G_{cand}$)
$G_{sub} \leftarrow \varnothing$ {set of gates to induce $C_{sub}$}
$G_{sub} \leftarrow G_{sub} \cup \{g_1, g_2\}$
**return** $G_{sub}$
```

There were two motivations for developing `RandomTwoGates`. We wanted to continue testing the capabilities of CORGI to determine if the selection/replacement process will work for C_{rep} with more than one output. We also had the intuition that a replacement for a multi-input, multi-output subcircuit would introduce new *external* control flows.

`RandomTwoGates` is (almost) purely a random selection algorithm. The only caveat is that not every pair of gates $(g_1, g_2) \subset G(C'_i)$ are “legal” selections since some pairs introduce cycles when replaced. Despite the fact that the candidates for selecting g_2 is a subset of $G(C'_i) - \{g_1\}$, `RandomTwoGates` is in no way a *smart selection* algorithm.

There were three results from analyzing the behavior of `RandomTwoGates`. First, we confirmed our intuition that new external control flows can indeed be introduced. Second, similar to `RandomSingleGate`, the circuit becomes very tall, with few gates in any single hierarchical level. Third, `RandomTwoGates` runs slower than

`RandomSingleGate` because CXL must select from a larger store of semantically equivalent replacements as the number of inputs increases. We will discuss each result separately.

By far the most profound discovery was that new external (and internal) control flows *can* be introduced (but it does not always occur). The reason it can is because the subcircuit selected can be (and often is) comprised of two gates, g_1 and g_2, which are not adjacent to each other (i.e., g_1 is not a predecessor of g_2). If g_1 and g_2 *are* adjacent, then the resulting subcircuit C_{sub} will have only one output, and `RandomTwoGates` will behave like `RandomSingleGate` for that single iteration.

The probability P that `RandomTwoGates` creates a new control flow during any given replacement iteration i is described by

$$P(i) \propto \left(1 - \frac{n_e}{X \times Y}\right) \times p_c \times p_a \tag{4.1}$$

where i, n_e, X, Y, p_c, and p_a are described below:

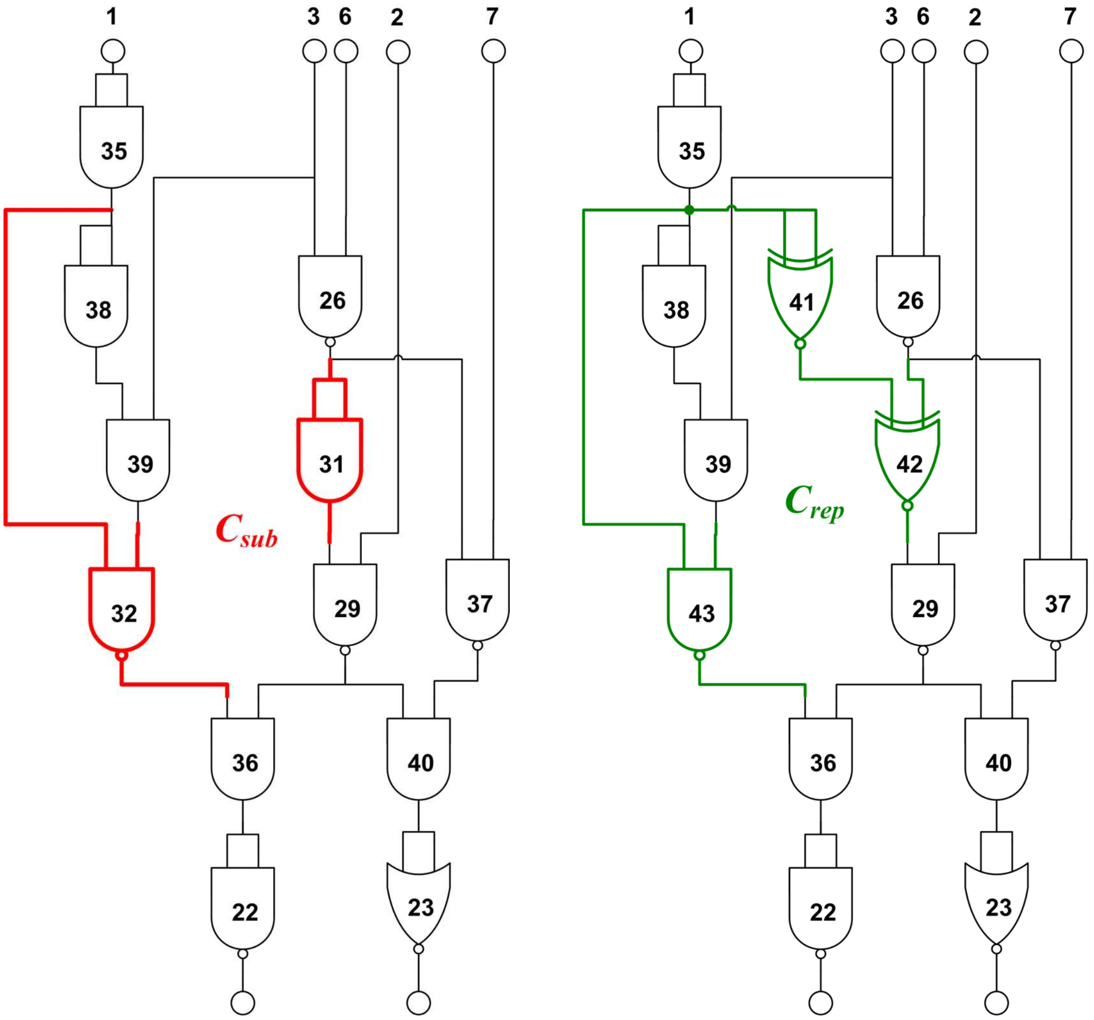
i	A particular iteration of the algorithm
n_e	Number of external control flows in C'_i before selection
X, Y	Number of inputs and outputs, respectively, of C'_i
p_c	Probability that CXL returns a replacement subcircuit C_{rep} with more control flows than C_{sub} (ref. Figure 4.4 for an example)
p_a	Probability that `RandomTwoGates` will choose two gates adjacent to one another (i.e., the output of one gate feeds an input of the other—ref. Section 4.3.2)

The foregoing can best be demonstrated with an actual circuit. Figure 3.2 depicts ISCAS benchmark C17, which was the target circuit for an experiment to demonstrate how selection algorithm `RandomTwoGates` can introduce new external control flows. A series of 20-iteration trials were performed until the final circuit C' had more external control flows than the original C (i.e., ISCAS benchmark C17). After only seven trials, a C' was found with a path from input 1 to output 23, which was not present in C. CORGI has the capability to output the results of each iteration of randomization, and by looking back through the data, we found that the seventh iteration produced the desired effect. Figure 4.2 shows the transition from C'_6 to C'_7 (iteration #7 in this example).

The second result for `RandomTwoGates`—the fact that it also makes circuits grow very tall—was somewhat unexpected. In retrospect, it probably should not have been since the same relationship between the hierarchical levels of C_{sub} and C_{rep} described in Section 4.3.2 exists for `RandomTwoGates`. As circuit size increases, the probability that C_{sub} will have two hierarchical levels ($\ell = 2$) decreases since the number of *adjacent* gate pairs in C'_i is exponentially smaller than the number of *all* gate pairs in C'_i. The rate at which a circuit obfuscated with `RandomTwoGates` grows taller is, on average, slightly slower than for `RandomSingleGate` since there is a non-zero probability that a one-output C_{sub} is selected during a given iteration of subcircuit selection and replacement.

The third result has to do with a non-intuitive property of circuit families. The size of a given family δ is a function of several factors, including input quantity, output quantity, basis, and gate quantity. But it is also a function of the signature (truth table) of elements of δ. Some families have circuit signatures such that there are relatively few (sometimes zero) elements. Others families may have thousands of elements. When a subcircuit C_{sub} is selected such that $|\delta|$ is large, the selection of a replacement C_{rep} takes longer. `RandomTwoGates` selects C_{sub} such that $\delta_{C_{sub}}$ (from which CXL must choose a C_{rep}) is, on average, larger than it is when `RandomSingleGate` is the selection algorithm. See [10] for more details on the relationship of a circuit's signature (truth table) to the size of its circuit family.

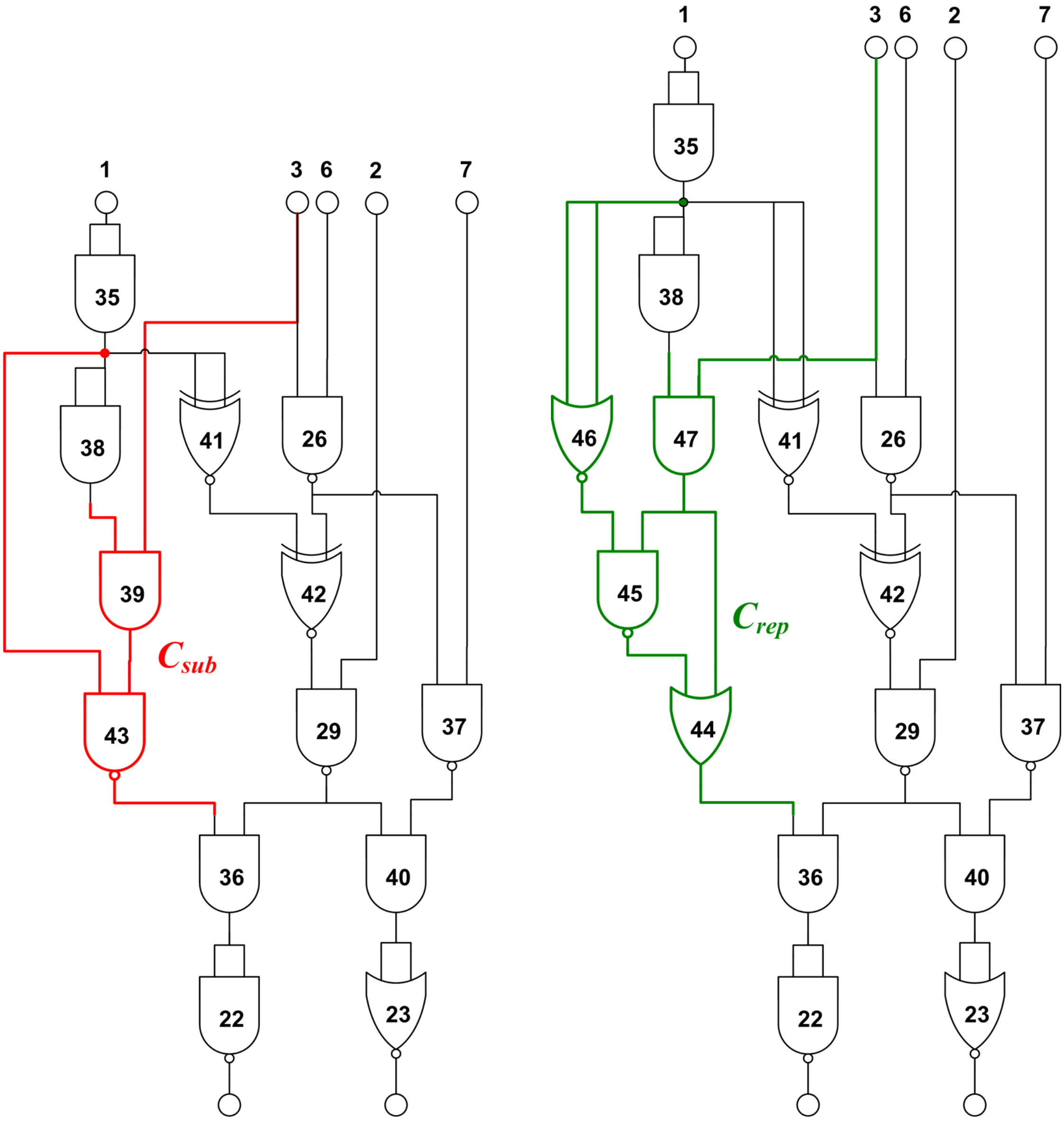
Reference Figure A.6(b) for a sample result of applying this algorithm to ISCAS benchmark circuit C17. Again, from the data presented in Section A.3, `RandomTwoGates` produces C' which are, on average, about half the height of circuits produced by `RandomSingleGate`.



(a) Circuit C'_6 with C_{sub} selected

(b) New circuit C'_7 with C_{rep} inserted

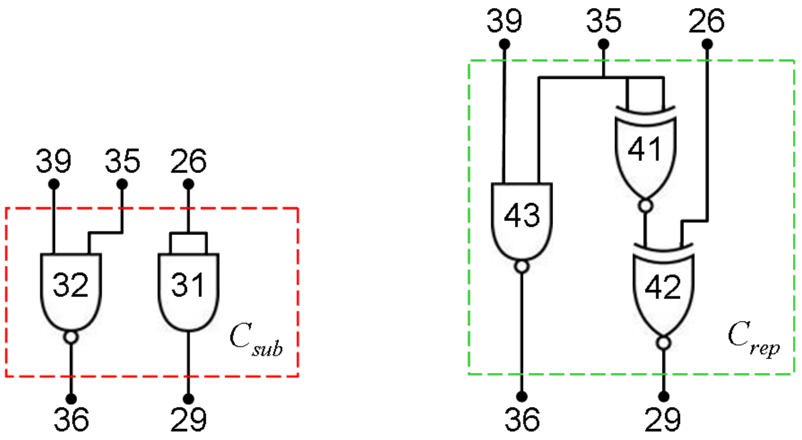
Figure 4.2: Subcircuit selection and replacement using `RandomTwoGates` on ISCAS C17, which creates a new external control flow in the circuit (input 1 [In1] to output 23 [Out23]).
(a) Gates 31 and 32 (C_{sub}) will be removed from C'_6. Note there is no control flow from In1 to Out23.
(b) New circuit C'_7 is created after C_{sub} in circuit C'_6 is replaced with semantically equivalent C_{rep} (gates 41, 42, and 43). A new control flow now exists from In1 to Out23 (path: In1→35→41→42→29→40→Out23).



(a) Circuit C'_7 with C_{sub} selected

(b) New circuit C'_8 with C_{rep} inserted

Figure 4.3: Subcircuit selection and replacement using `RandomTwoGates` on ISCAS C17, which replaces a two gates, each added during different iterations.
(a) Gates 39 and 43 (C_{sub}) will be removed from C'_7. Note that gate 39 was not in the original circuit.
(b) New circuit C'_8 is created after C_{sub} in circuit C'_7 is replaced with semantically equivalent C_{rep} (gates 44, 45, 46, and 47). Because of the structure of the selected replacement circuit, gates 38 and 35 are each elevated to the next higher layer in the circuit hierarchy.



(a) C_{sub} from Figure 4.2(a)

(b) C_{rep} from Figure 4.2(b)

Inputs			*Outputs*	
26	**35**	**39**	**31/42**	**32/43**
F	F	F	F	T
F	F	T	F	T
F	T	F	F	T
F	T	T	F	F
T	F	F	T	T
T	F	T	T	T
T	T	F	T	T
T	T	T	T	F

(c) Truth table of C_{sub} and C_{rep}

Figure 4.4: An example of how a replacement subcircuit C_{rep} can introduce a new control flow where none existed in the selected subcircuit C_{sub} (reference Figure 4.2).
(a) No control flow exists between gate 35 and gate 29 in C_{sub}.
(b) Subcircuit C_{rep} has a control flow from gate 35 to gate 29.
(c) The truth table of both circuits.

4.3.4 ***`RandomLevelTwoGates`.*** The `RandomLevelTwoGates` selection algorithm functions the same as `RandomTwoGates` except that G_{cand} only contains gates which are in at most three contiguous levels of the circuit hierarchy. All subcircuit selections C_{sub} are of the class $C_{X_{sub}\text{-}Y_{sub}\text{-}2\text{-}\Omega_{sub}}$ where

$$\Omega_{sub} \subset \{\text{AND, NAND, OR, NOR, XOR, XNOR}\}$$
$$X_{sub} \geq 2$$
$$Y_{sub} = 1 \text{ or } Y_{sub} = 2$$
$$S_{sub} = 2$$
$$|\Omega_{sub}| = 1 \text{ or } |\Omega_{sub}| = 2$$
$$\ell_{g_2} = \ell_{g_1} \pm 1$$

The *similarity* between `RandomLevelTwoGates` and `RandomTwoGates` is in how it selects the first gate, g_1. In both cases, g_1 is selected entirely at random. Since gate g_1 occupies some hierarchical level ℓ_{g_1}, then level ℓ_{g_1} is a de facto random selection. The *difference* between these two algorithms is in how gate g_2 is selected. With `RandomLevelTwoGates`, gate g_2 must be selected from within levels ℓ_{g_1}, $\ell_{g_1} + 1$, or $\ell_{g_1} - 1$. As we saw with `RandomTwoGates`, its gate g_2 selection can be any gate that does not introduce a cycle. Note that `RandomLevelTwoGates` selects two gates which are within one level of each other. Therefore, no call to `RejectGates` is required since it is impossible to introduce a cycle.

Having observed that `RandomTwoGates` increases the number of hierarchical levels (i.e., the *height*) at approximately half the rate of `RandomSingleGate` (see Section A.3 for discussion), we developed `RandomLevelTwoGates` to see if we could further reduce the rate of height increase relative to the number of iterations performed, yet retain as much randomness as possible otherwise. The conjecture was that, by limiting subcircuit selection to gates in a single “band” of at most three hierarchical levels,

Algorithm 3 `RandomLevelTwoGates`(C'_i)

```
call EstablishGateHierarchy() {Assigns each gate to a hierarchical level}
$G_{cand} \leftarrow \varnothing$
$g_1 \leftarrow$ call SelectRandomGate($G(C'_i)$) {A random gate from any level}
$\ell_{g_1} \leftarrow$ hierarchy level of gate $g_1$
$G_{cand} \leftarrow G_{cand} \cup G(\ell_{g_1})$ {all gates in level $\ell_{g_1}$}
$G_{cand} \leftarrow G_{cand} - \{g_1\}$
if $\ell_{g_1} > 0$ then
    $G_{cand} \leftarrow G_{cand} \cup G(\ell_{g_1} - 1)$ {all gates one level below $g_1$}
end if
if $\ell_{g_1} < \ell_{MAX}$ then
    $G_{cand} \leftarrow G_{cand} \cup G(\ell_{g_1} + 1)$ {all gates one level above $g_1$}
end if
$g_2 \leftarrow$ call SelectRandomGate($G_{cand}$)
$G_{sub} \leftarrow \varnothing$
$G_{sub} \leftarrow G_{sub} \cup \{g_1, g_2\}$
return $G_{sub}$
```

the propensity of a replacement circuit C_{rep} to increase the circuit's height should be further mitigated.

The fact that we specifically disregard particular levels when choosing g_2 makes `RandomLevelTwoGates` a *smart selection* algorithm. Gate g_1 is still selected randomly, but since the subset of gates from which g_2 is chosen is dependent on g_1, we expect to be able to better control how `RandomLevelTwoGates` modifies a circuit. The reason that g_2 is not restricted *only* to ℓ_{g_1} is because of the nature of subcircuits C_{rep} returned by CXL. As previously discussed in Section 4.3.2 (page 38), C_{rep} can—and often does—have more than one hierarchical level. If it occurs that C_{sub} has the same height as C_{rep}, then the overall circuit C'_i will not grow in height during iteration i.

There were four results from analyzing the behavior of `RandomLevelTwoGates`. First, we confirmed our hypothesis that `RandomLevelTwoGates` produces shorter circuits than either `RandomSingleGate` or `RandomTwoGates`. Second, we demonstrated that a smart selection strategy can be employed to guide the behavior of a white-box obfuscator to a particular end. In this case, we took our observations of how single iterations of random selection strategies impacted circuit growth to develop a smart algorithm.

The third result is the nature of the internal circuit structure. Unlike the two random selection algorithms, `RandomLevelTwoGates` has connections (edges) which span fewer hierarchical levels. This can be observed by comparing Figures A.5(b) and A.6(b). In the former image, many connections span more that half the length of the circuit, whereas in the latter image, connections spanning more that eight levels appear much less frequently. The implication of this finding is that level-based algorithms could be useful if connection length is a circuit property that correlates to the degree of obfuscation.

The fourth result has to do with algorithm efficiency. The function `EstablishGateHierarchy` is a component of this (indeed, all four) level-based algorithm. It must be invoked at the beginning of every iteration, as shown in algorithms 3, 4 , 5, and 6 (line 1 in each). In its current implementation, `EstablishGateHierarchy` is inefficient.[1] For C_i' with small size $|S|$, this is not a problem; but as the number of iterations increase, the circuit size also increases, and `EstablishGateHierarchy` slows down the selection algorithm. Future versions of CORGI must take this performance factor into account in order that level-based selection algorithms are efficient for large circuits.

4.3.5 FixedLevelTwoGates.

The `FixedLevelTwoGates` selection algorithm functions the same as `RandomLevelTwoGates` except for two differences. Whereas in `RandomLevelTwoGates` the target level is based on the selection of gate g_1, the opposite is true here. `FixedLevelTwoGates` must first have a level ℓ_F to target (user input), and from that level, it selects gate g_1 (the numerical value of ℓ_F remains constant for all iterations). In addition, `FixedLevelTwoGates` selects gate g_2 only from levels ℓ_F or $\ell_F + 1$ (not from $\ell_F - 1$). All subcircuit selections C_{sub} are of the class $C_{X_{sub}\text{-}Y_{sub}\text{-}2\text{-}\Omega_{sub}}$

---

[1] The details of why this is the case are discussed in Appendix A

where

$$\Omega_{sub} \subset \{\text{AND, NAND, OR, NOR, XOR, XNOR}\}$$

$$X_{sub} \geq 2$$

$$Y_{sub} = 1 \text{ or } Y_{sub} = 2$$

$$S_{sub} = 2$$

$$|\Omega_{sub}| = 1 \text{ or } |\Omega_{sub}| = 2$$

$$\ell_{g_2} = \ell_{g_1} \text{ or } \ell_{g_2} = \ell_{g_1} + 1$$

Algorithm 4 `FixedLevelTwoGates`(C'_i)

```
call EstablishGateHierarchy()
$G_{cand} \leftarrow \varnothing$
$\ell_F \leftarrow$ fixed level where $0 \leq F \leq \ell_{MAX}$ {user inputs $F$}
$G_{cand} \leftarrow G_{cand} \cup G(\ell_F)$ {all gates in level $\ell_F$}
$g_1 \leftarrow$ call SelectRandomGate($G_{cand}$)
$G_{cand} \leftarrow G_{cand} - \{g_1\}$
if $\ell_F < \ell_{MAX}$ then
    $G_{cand} \leftarrow G_{cand} \cup G(\ell_{g_1} + 1)$
end if
$g_2 \leftarrow$ call SelectRandomGate($G_{cand}$)
$G_{sub} \leftarrow \varnothing$
$G_{sub} \leftarrow G_{sub} \cup \{g_1, g_2\}$
return $G_{sub}$
```

The first three algorithms developed successively improved control over circuit growth as measured by circuit height, yet they each created a wide range of height results. In other words, over many trials, the data shows a large standard deviation (σ) for circuit height (see Figure A.2). Our motivation for developing `FixedLevelTwoGates` next was to observe whether targeting a single level for subcircuit selection would cause the circuit to grow wider than it did with `RandomLevelTwoGates`.

There were two findings regarding `FixedLevelTwoGates`, one negative, and one positive. First, it produces circuits which are (on average) both taller and narrower than those produced by `RandomLevelTwoGates`. This is the opposite of what we expected, but the circuits did exhibit one similarity to those produced by `RandomLevelTwoGates`; namely, there are relatively few connections that span more than 10% of the circuit's height.

Second, however, `FixedLevelTwoGates` achieved more predictable behavior relative to the number of iterations performed (i.e., smaller standard deviation, σ). We attribute that fact to limiting the selection of gate g_2 to only two, rather than three, contiguous levels in C_i'. This substantially limits the set of gates from which gate g_1 may be selected (the previous three algorithms select gate g_1 at random from among *all* gates in C_i'). As a result, this *smart* selection algorithm has much less randomness, which may be the basis of the tight coupling between circuit height and number of iterations.

4.3.6 `LargestLevelTwoGates`. With `LargestLevelTwoGates`, we combine the *variable* level selection of `RandomLevelTwoGates` with the *targeted* level selection of `FixedLevelTwoGates`. This algorithm is procedurally the same as `FixedLevelTwoGates` except that the selected largest (widest) level, ℓ_W, is calculated for every iteration.

Algorithm 5 `LargestLevelTwoGates`(C_i')

```
call EstablishGateHierarchy()
$G_{cand} \leftarrow \varnothing$
$\ell_W \leftarrow$ widest (largest) level where $0 \le \ell_W \le \ell_{MAX}$ {tiebreaker: smallest $\ell_W$}
$G_{cand} \leftarrow G_{cand} \cup G(\ell_W)$ {all gates in level $\ell_W$}
$g_1 \leftarrow$ call SelectRandomGate($G_{cand}$)
$G_{cand} \leftarrow G_{cand} - \{g_1\}$
if $\ell_W < \ell_{MAX}$ then
    $G_{cand} \leftarrow G_{cand} \cup G(\ell_{g_1} + 1)$
end if
$g_2 \leftarrow$ call SelectRandomGate($G_{cand}$)
$G_{sub} \leftarrow \varnothing$
$G_{sub} \leftarrow G_{sub} \cup \{g_1, g_2\}$
return $G_{sub}$
```

Our objective in developing `LargestLevelTwoGates` is an algorithm that is agile enough to "chase" the largest (widest) level as C_i' grows. The nature of subcircuit replacement, combined with the rigidity of predecessor relationships in a combinatorial Boolean circuit, causes gates to migrate to higher levels in the circuit hierarchy. When a gate moves from one level to the next, the population of the level it originally occupied decrements by one. To combat this tendency, `LargestLevelTwoGates` always selects gates from the largest level. If multiple levels are largest, choose the lowest level among them.

Two results from `LargestLevelTwoGates` are clearly evident in Figure A.7(c). First, the algorithm provides more control over circuit growth than any of the previous selection algorithms. From the data in Section A.3, we see the average height of C' produced by `LargestLevelTwoGates` is approximately 54% the average height of C' produced by its nearest competitor, `RandomLevelTwoGates`. Circuits produced using `LargestLevelTwoGates` are also much wider than any of the other circuits. `RandomLevelTwoGates` is again the closest competition, but `LargestLevelTwoGates` produces C' more than twice as wide.

A second result is the fact that `LargestLevelTwoGates` can introduce external control flows, just as we first saw with `RandomTwoGates`. The C' circuit represented in Figure A.7(c) has external control flows In1–Out23 and In7–Out22, whereas the original circuit C in Figure A.7(a) does not. Thus, `LargestLevelTwoGates` provides a high degree of control over circuit growth, yet retains the potential to introduce control flows.

4.3.7 `OutputLevelTwoGates`.

The last of the six algorithms is another variation on a theme. `OutputLevelTwoGates` is, in fact, only a special case of `FixedLevelTwoGates` where the target level contains the circuit outputs. In other words, using `FixedLevelTwoGates` with $\ell_F = 0$ is the same as using `OutputLevelTwoGates`. Algorithm 6 shows this special case.

Algorithm 6 `OutputLevelTwoGates`(C'_i)

```
call EstablishGateHierarchy()
$G_{cand} \leftarrow \varnothing$
$\ell_0 \leftarrow$ level 0 which only contains circuit output gates
$G_{cand} \leftarrow G_{cand} \cup G(\ell_0)$ {all gates in level $\ell_0$}
$g_1 \leftarrow$ call SelectRandomGate($G_{cand}$)
$G_{cand} \leftarrow G_{cand} - \{g_1\}$
$G_{cand} \leftarrow G_{cand} \cup G(\ell_1)$ {all gates in level $\ell_1$}
$g_2 \leftarrow$ call SelectRandomGate($G_{cand}$)
$G_{sub} \leftarrow \varnothing$
$G_{sub} \leftarrow G_{sub} \cup \{g_1, g_2\}$
return $G_{sub}$
```

`OutputLevelTwoGates` was developed purely out of curiosity, and it proved to be a worthwhile endeavor. When $\ell_F > 0$ in `FixedLevelTwoGates`, there is the possibility that the width of ℓ_F can increase. Conversely, the width of a circuit's output level (ℓ_0) is fixed, so any replacement of a subcircuit that contains an output gate must not increase the number of circuit outputs.[2] The net effect on circuit growth is best described by way of analogy, followed by three example C' circuits produced by `OutputLevelTwoGates`.

The behavior of `OutputLevelTwoGates` resembles the manufacturing process of extrusion which creates long objects of a fixed cross-sectional profile. In this case, the cross-sectional profile is circuit width. However, unlike the random algorithms, `OutputLevelTwoGates` produces circuits in which *all*[3] levels are approximately the same width as output level ℓ_0. Regardless of how many outputs the circuit has, or how many iterations are performed, the widest level will contain only a few more gates than the output level, ℓ_0.

To see the extrusion effect, reference C' in Figure A.5(c) which has height of 93 levels and widest layer only of 4 gates. But on average, all layers are not 4

---

[2] Under the concept of *black-box refinement*, adding decoy outputs—and inputs—is desireable. This research does not address the concept, however, so we restrict ourselves to preserving circuit input and output quantities.

[3] An exception to this is when, prior to iteration 1, the widest level of C is substantially wider than ℓ_0.

gates wide; they are only 2 gates wide, which is equal to the number of outputs. For another demonstration, we apply `OutputLevelTwoGates` to a different ISCAS benchmark circuit, C880, which has 26 outputs. The results, for different numbers of iterations, are shown in Figures A.8–A.10. Again, the average width of all layers is approximately 26, which is the same as ℓ_0.

From these results, we must revisit our observations for `FixedLevelTwoGates`. Basically, `FixedLevelTwoGates` behaves the same as `OutputLevelTwoGates`, but the extrusion occurs at some user-defined level. In essence, the target circuit C will be "split" at the chosen level ℓ_F which has a particular number of gates, n_F. All levels 0 through $\ell_F - 1$ will remain unchanged and an extruded subcircuit will connect the top and bottom of C in the final randomized circuit C'.

4.4 Runtime performance analysis

We conclude with a brief discussion on the runtime performance of the six algorithms. This is not intended to be a rigorous examination of CORGI performance, but instead it will provide an understanding of what factors influence run times as well as compare the performance of each of the six selection algorithms relative to one another. Figures 4.5–4.16 contain runtime performance data for the six algorithms as applied to two different ISCAS BENCH circuits: C17 and c880. Each figure displays representative results from two trials for each combination (i.e., selection algorithm and circuit). In all cases, each trial is 1000 iterations. Table 4.1 provides a summary of the data.

The time required for each iteration of randomization is comprised of the time needed to perform the selection and the time required for CXL to produce a replacement subcircuit. For all algorithms, the latter is independent of both the structure of C'_i and the time required for CORGI to select a subcircuit from C'_i; however, the runtime of CXL is dependent on whether it generates an equivalent subcircuit at runtime, or simply selects an equivalent subcircuit from a static store. For the data presented here, we used the runtime option. Even though that choice is more time-

Algorithm	**C17**		**C880**	
	Trial 1	*Trial 2*	*Trial 1*	*Trial 2*
`RandomSingleGate`	38 ms	37 ms	44 ms	44 ms
`RandomTwoGates`	441 ms	447 ms	476 ms	504 ms
`RandomLevelTwoGates`	290 ms	282 ms	430 ms	436 ms
`FixedLevelTwoGates`	404 ms	393 ms	420 ms	438 ms
`LargestLevelTwoGates`	350 ms	359 ms	400 ms	373 ms
`OutputLevelTwoGates`	331 ms	359 ms	438 ms	445 ms

Table 4.1: Summary of runtime data for the six selection algorithms. The data show the average per-iteration time (in milliseconds) after 1000 iterations for two trials on each of two circuits: C17 and C880. Times are rounded to the nearest millisecond.

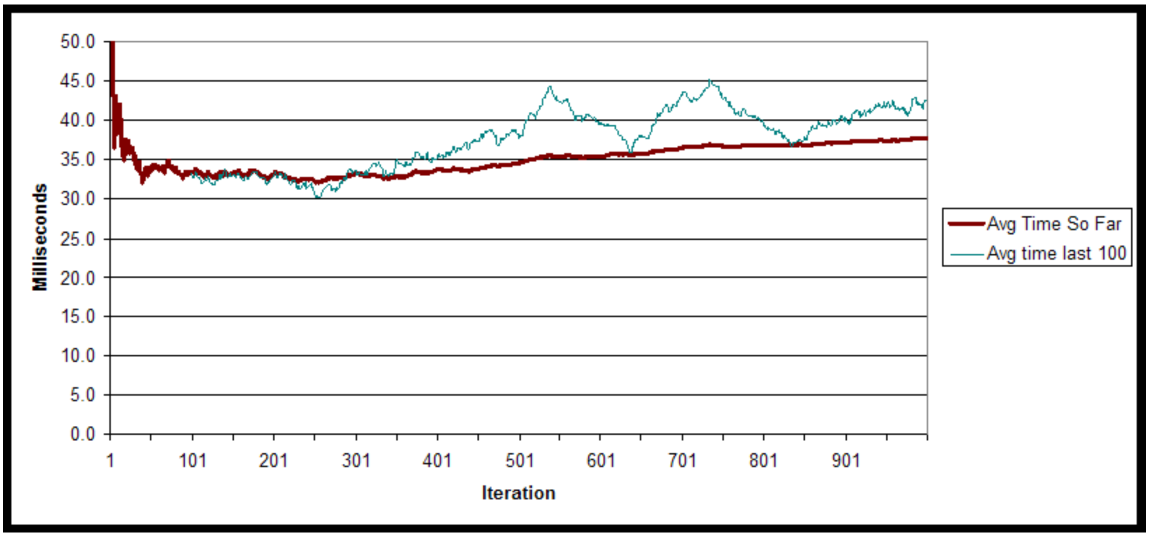
intensive, the average per-iteration time will remain constant over many iterations. Therefore, by comparing the results of one selection algorithm to those of another (or the same selection algorithm applied to different circuits), we can deduce the relative performance characteristics of CORGI.

`RandomSingleGate` is the fastest of the six selection algorithms. Each subcircuit only has one gate, thus the subcircuit only has one output. As a result, CXL can more quickly return a replacement. The slower times for `RandomSingleGate` when applied to C880 vs. C17 is because C880 initially has 437 gates to only 6 gates for C17.

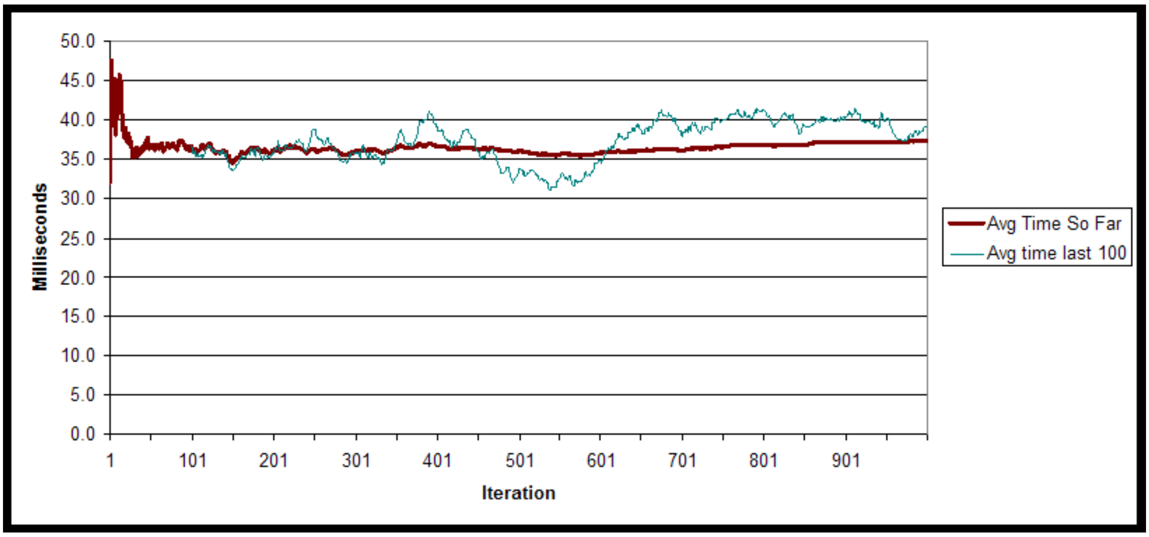
The remaining five selection algorithms are substantially slower than `RandomSingleGate` primarily because selected subcircuits contain two outputs. Therefore, the library of equivalent subcircuits in CXL is substantially larger. Since, for our experiments, CXL generates the equivalent subcircuits at runtime, the per-iteration times increase substantially as compared to `RandomSingleGate`.

`RandomTwoGates` is the slowest of the six selection algorithms. `RandomTwoGates` is the only selection algorithm that calls `RejectGates` (Algorithm 9), which is a recursive DFS.

For the four level-based selection algorithms, average run times over 1000 iterations are all less than times for `RandomTwoGates`. Whereas `RandomTwoGates` calls `RejectGates`, the four level-based selection algorithms all call `EstablishGateHierarchy`. This, too, is a DFS, but employs *pruning*. Pruning is a graph theory technique for limiting the search space, and in part accounts for the relative speedup of these four algorithms as compared to `RandomTwoGates`. Another factor that contributes to the increased speed of these four algorithms is the frequency of selecting single output subcircuits. Specifically, these algorithms select gates in adjacent hierarchical layers, which means the second gate selected by the algorithms is more likely to be a predecessor or successor of the first gate selected. The result of such a selection is a one-output subcircuit, which CXL more quickly produces than it does two-output subcircuits.

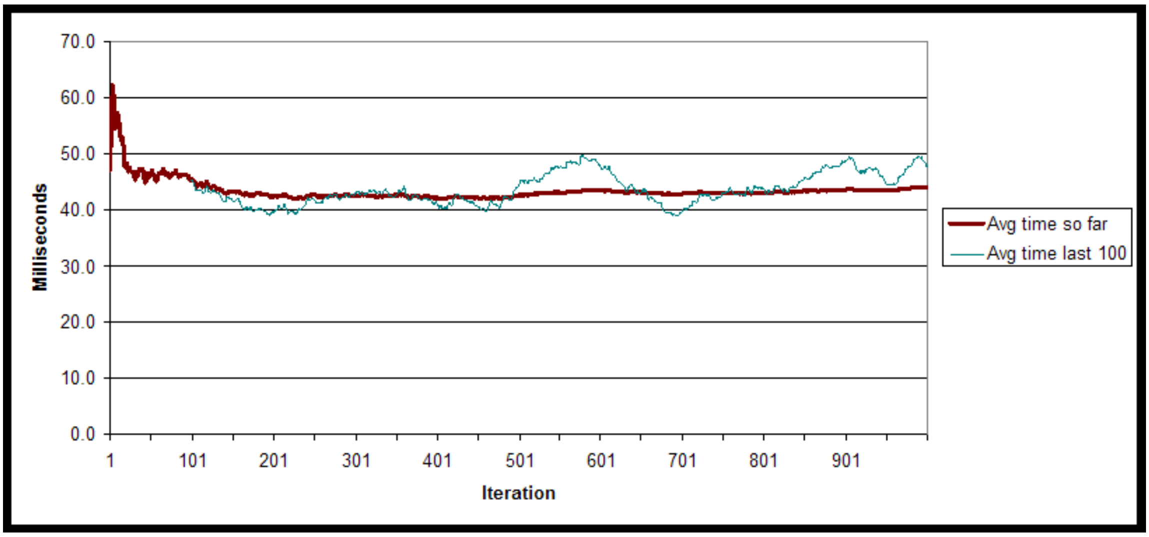


(a) Trial 1

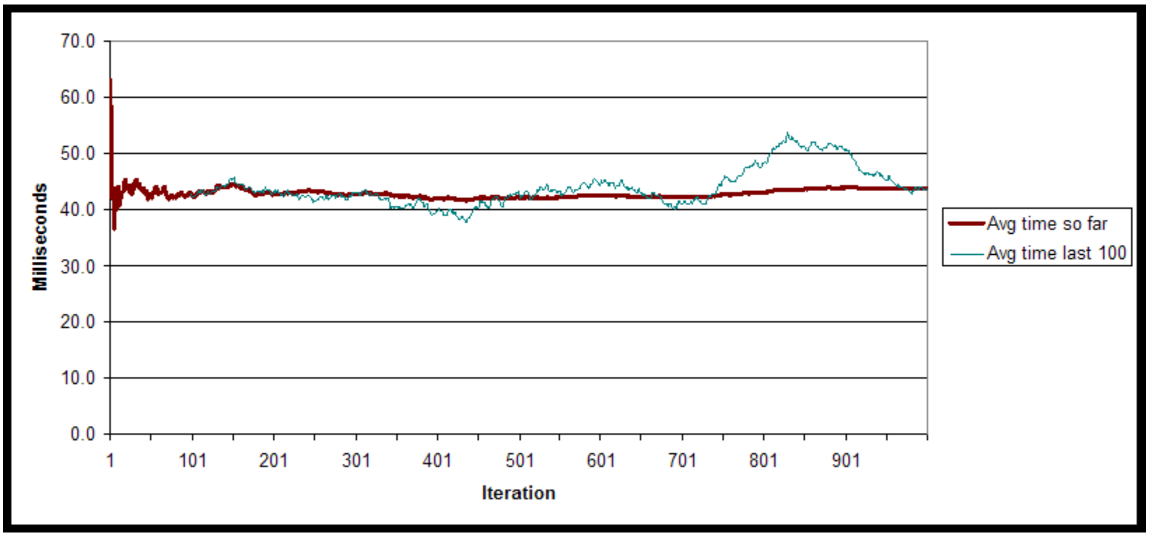


(b) Trial 2

Figure 4.5: Sample per-iteration runtime data from applying selection algorithm `RandomSingleGate` to circuit C17.

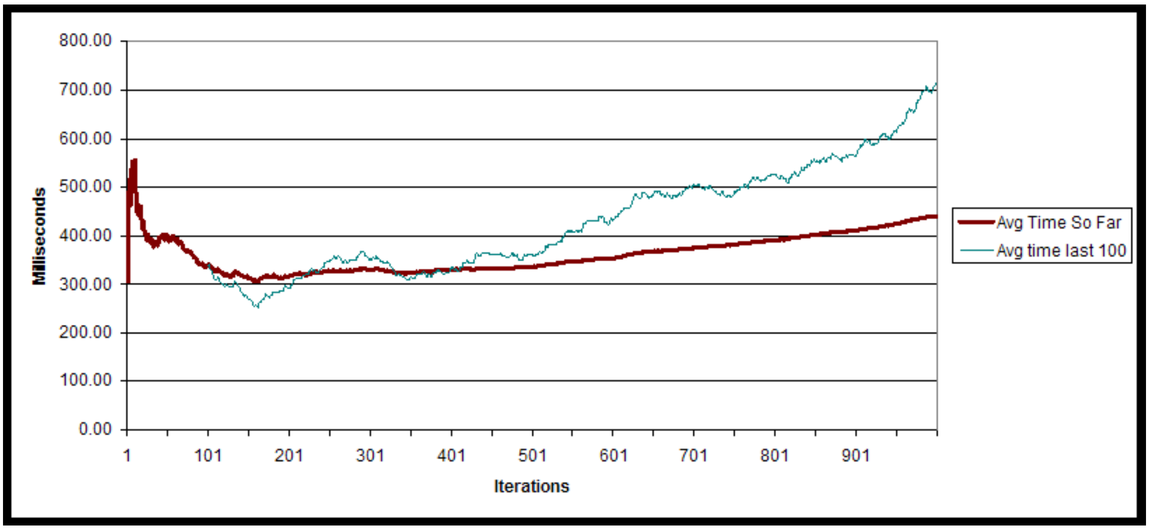


(a) Trial 1

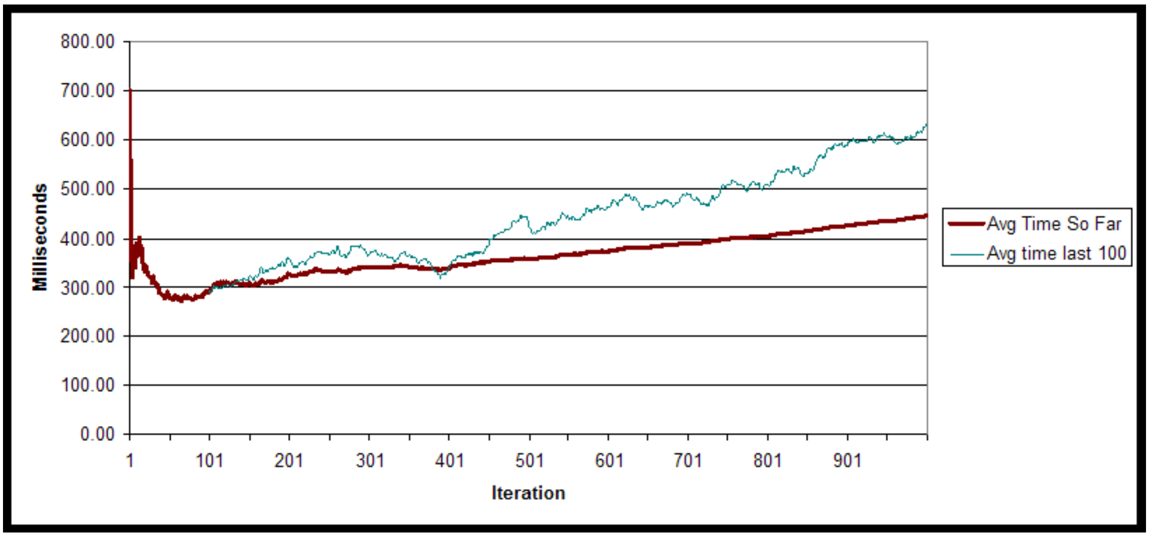


(b) Trial 2

Figure 4.6: Sample per-iteration runtime data from applying selection algorithm `RandomSingleGate` to circuit C880.

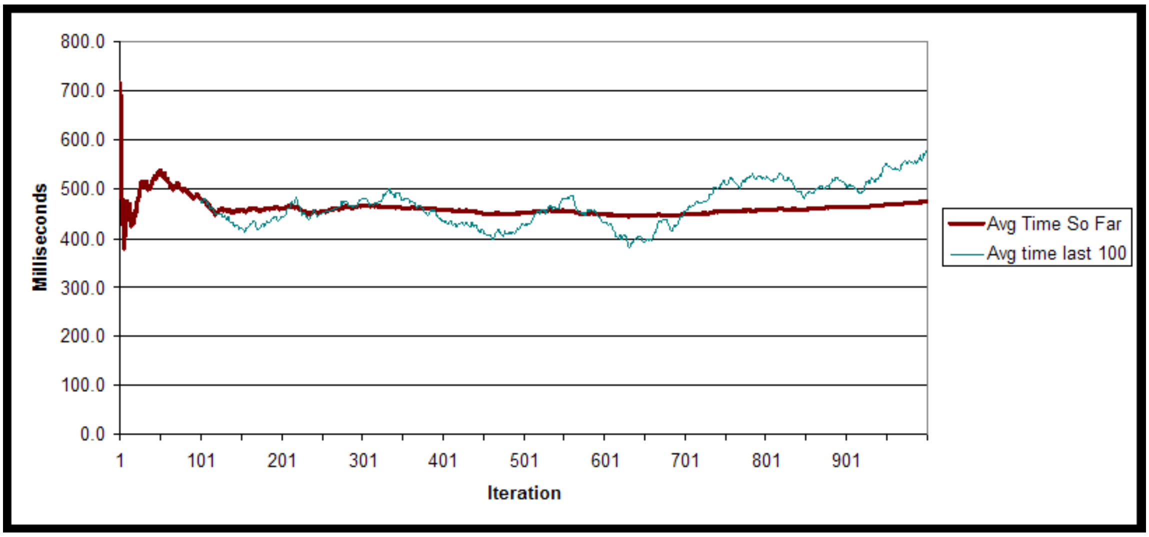


(a) Trial 1

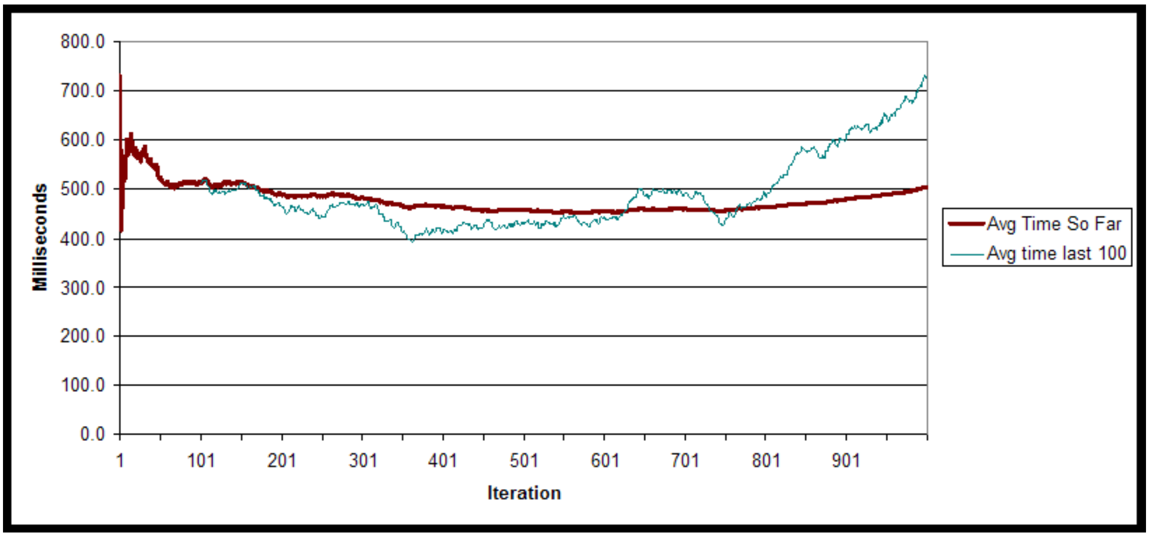


(b) Trial 2

Figure 4.7: Sample per-iteration runtime data from applying selection algorithm `RandomTwoGates` to circuit C17.

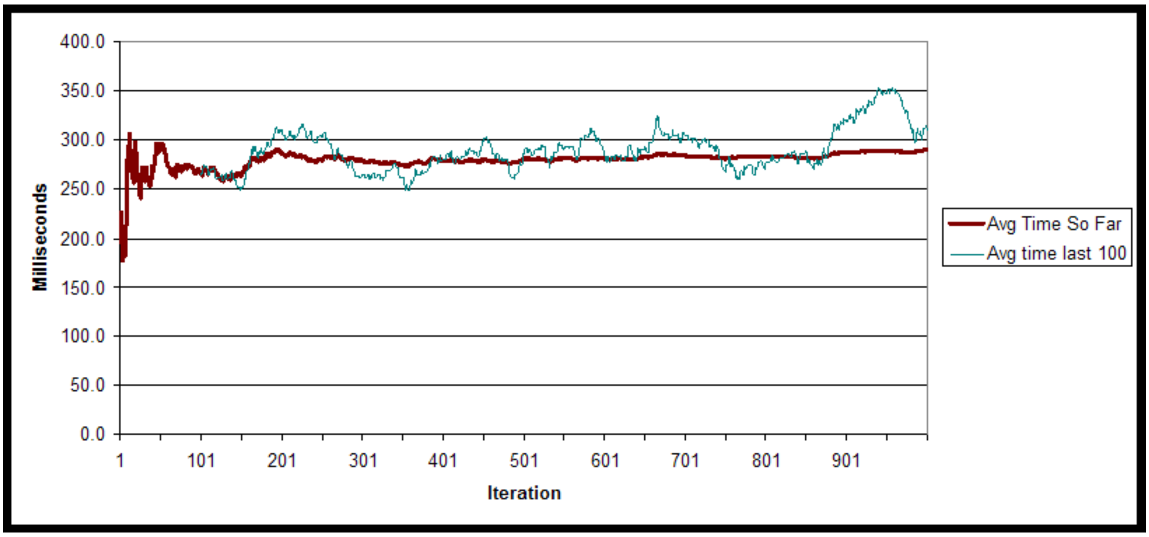


(a) Trial 1

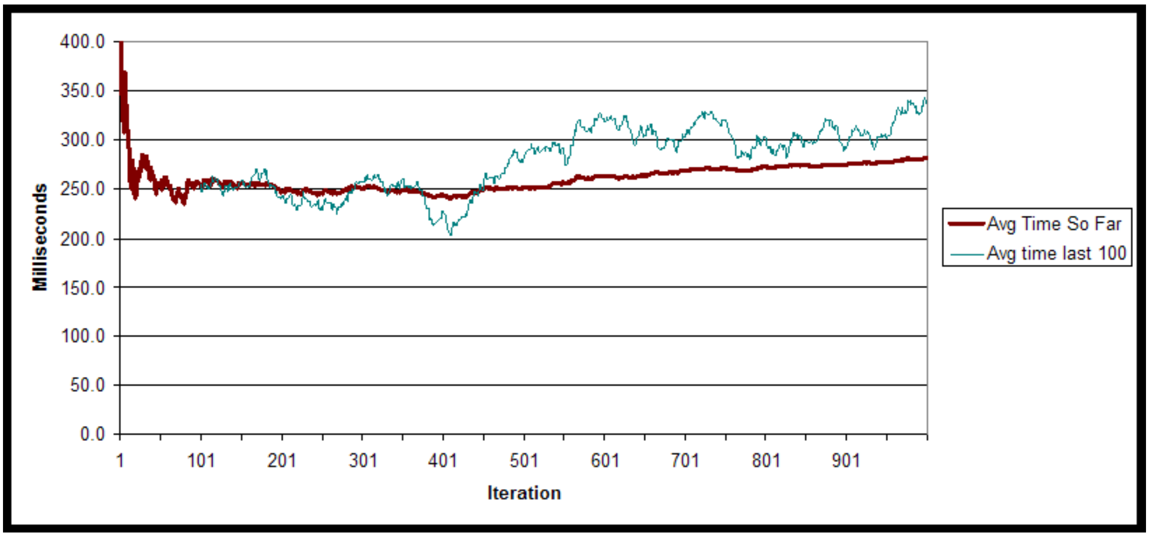


(b) Trial 2

Figure 4.8: Sample per-iteration runtime data from applying selection algorithm `RandomTwoGates` to circuit C880.

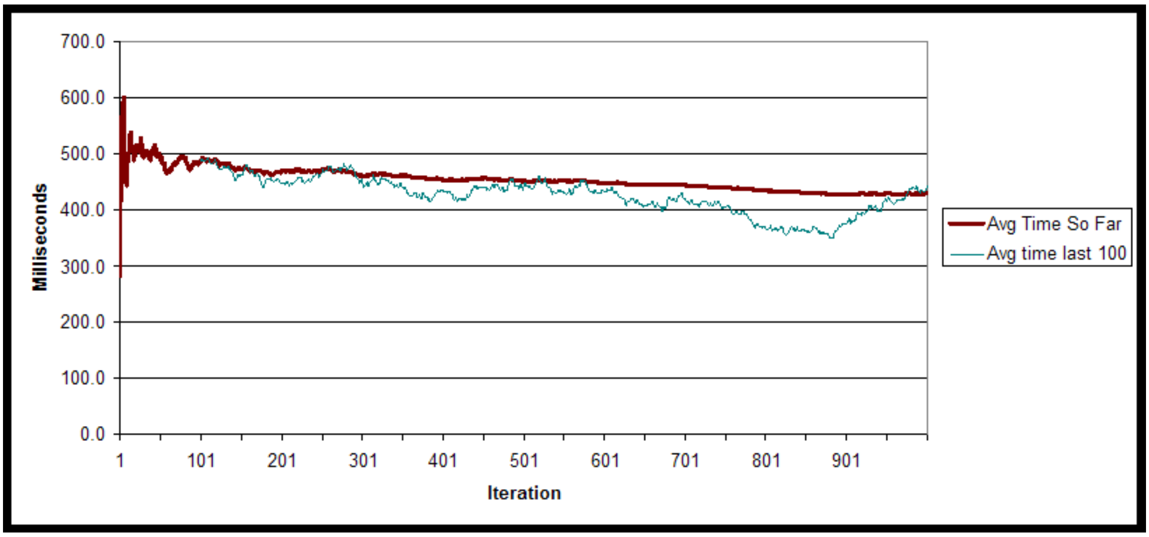


(a) Trial 1

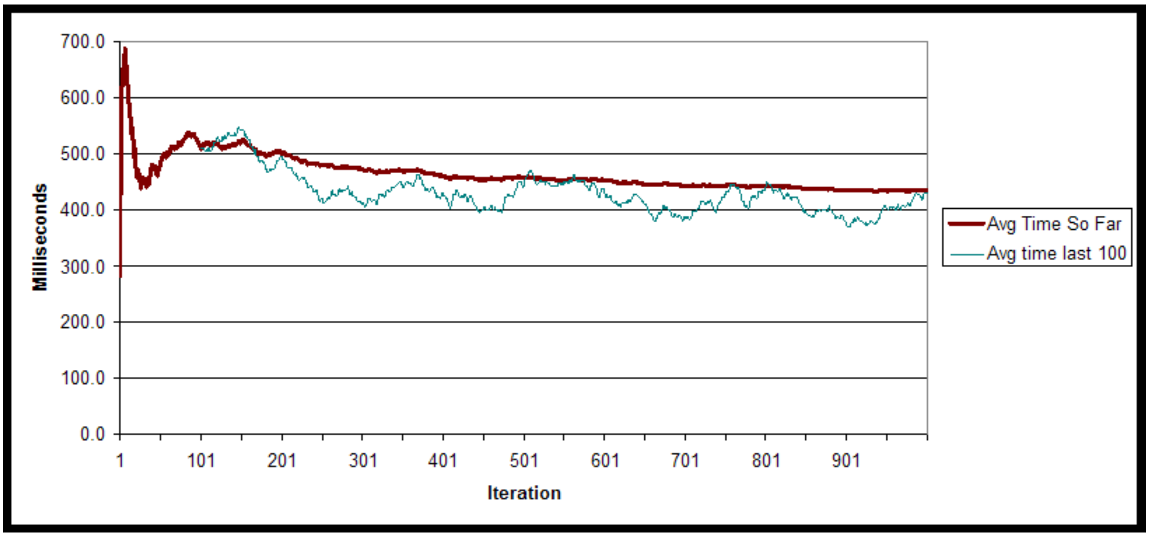


(b) Trial 2

Figure 4.9: Sample per-iteration runtime data from applying selection algorithm `RandomLevelTwoGates` to circuit C17.

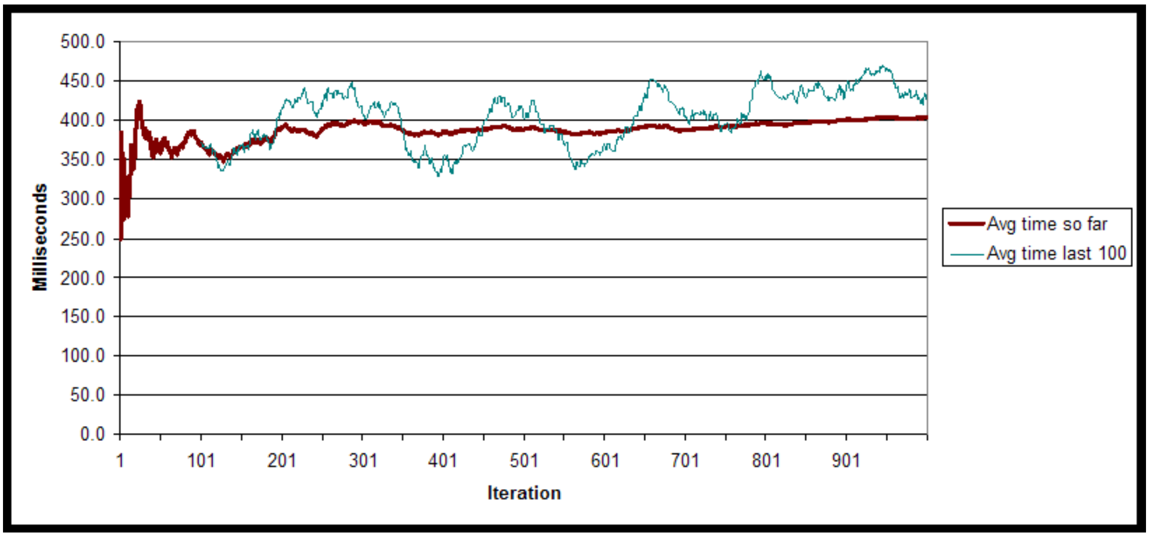


(a) Trial 1

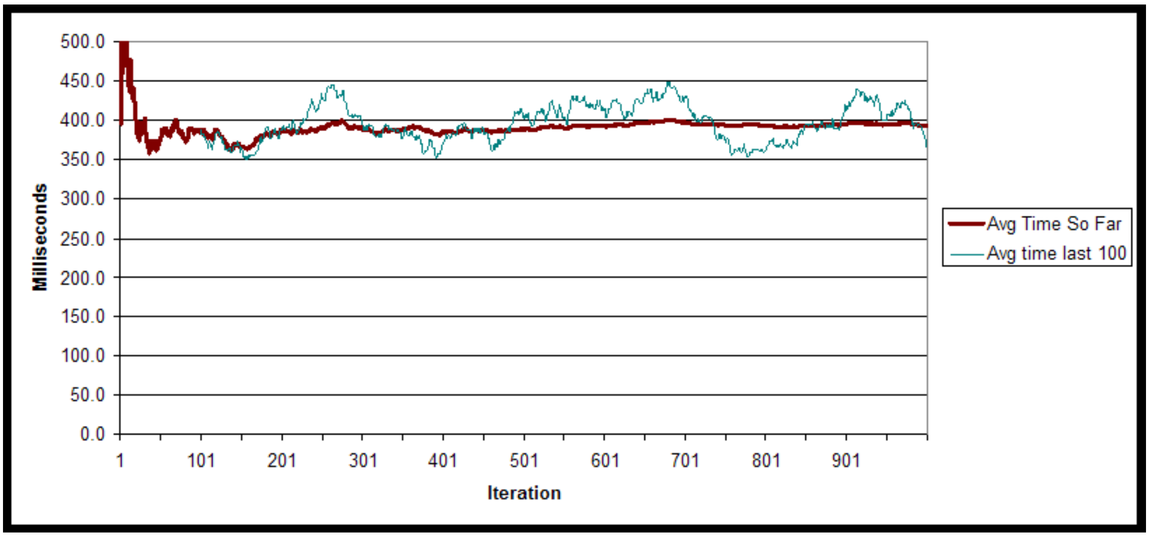


(b) Trial 2

Figure 4.10: Sample per-iteration runtime data from applying selection algorithm `RandomLevelTwoGates` to circuit C880.

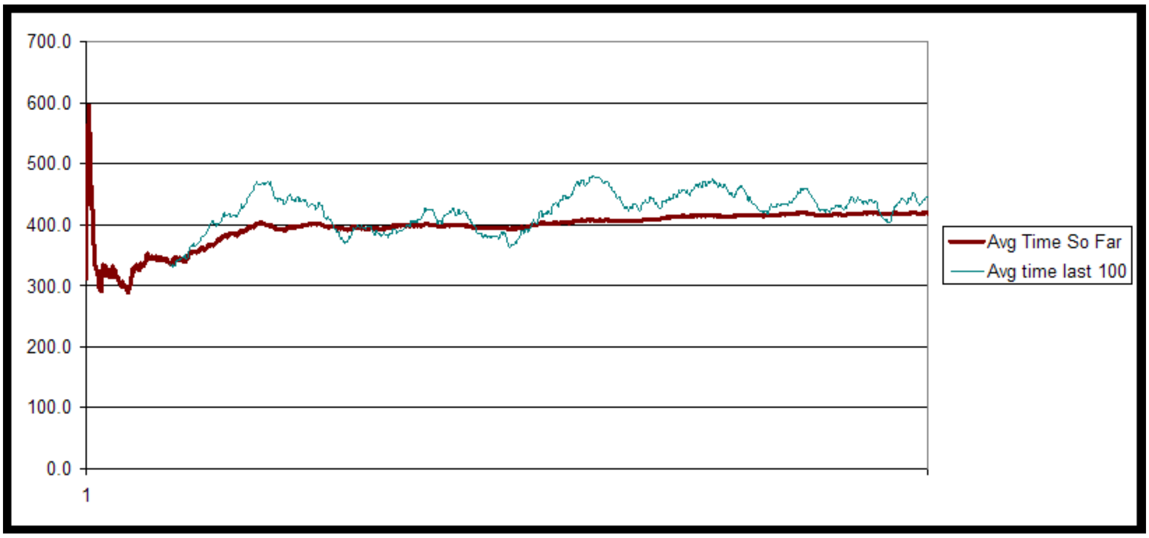


(a) Trial 1

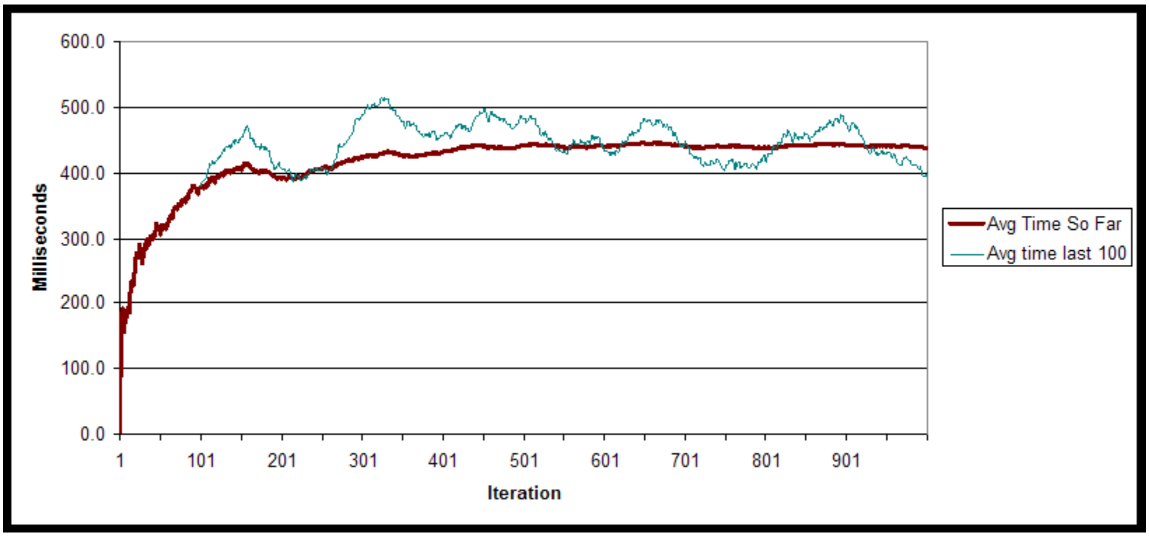


(b) Trial 2

Figure 4.11: Sample per-iteration runtime data from applying selection algorithm `FixedLevelTwoGates` to circuit C17.

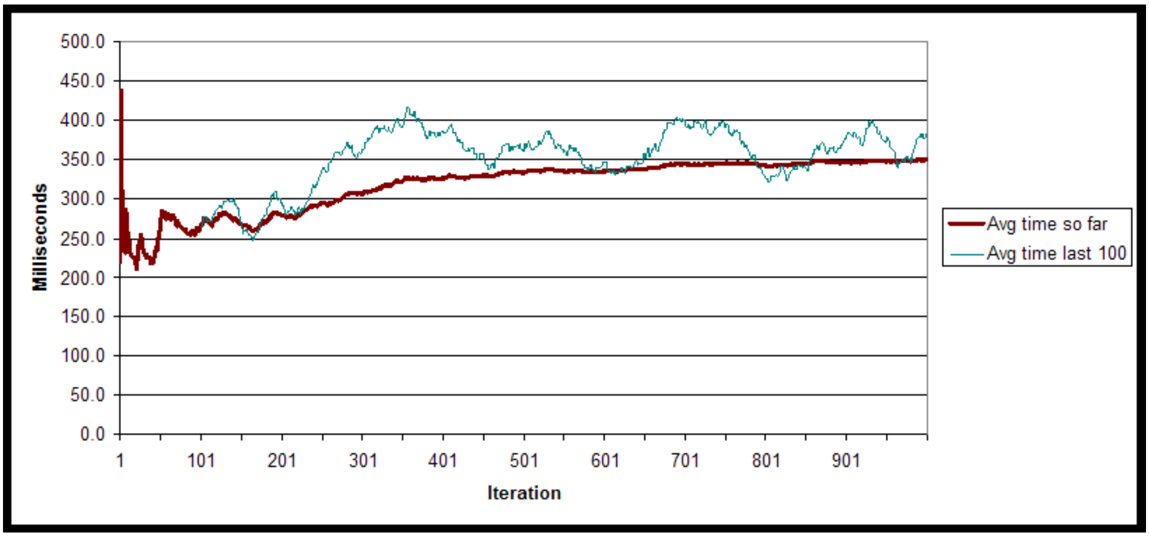


(a) Trial 1

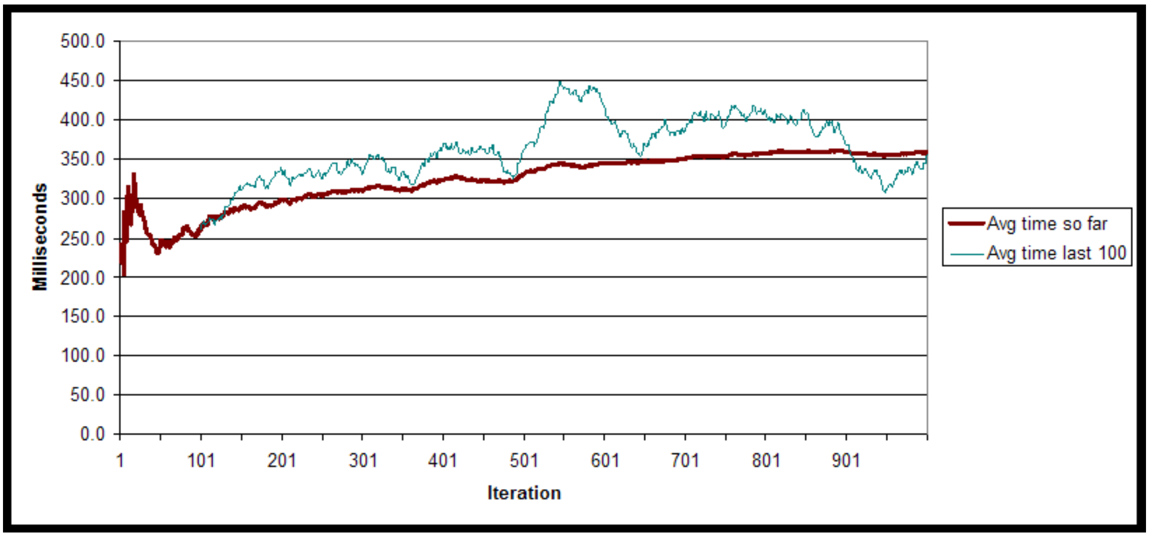


(b) Trial 2

Figure 4.12: Sample per-iteration runtime data from applying selection algorithm `FixedLevelTwoGates` to circuit C880.

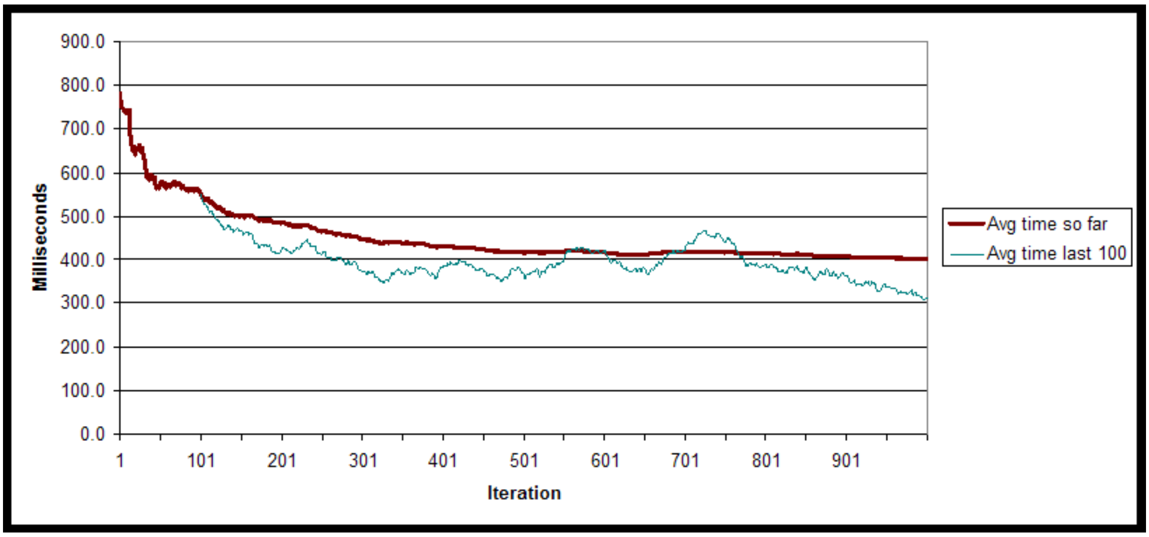


(a) Trial 1

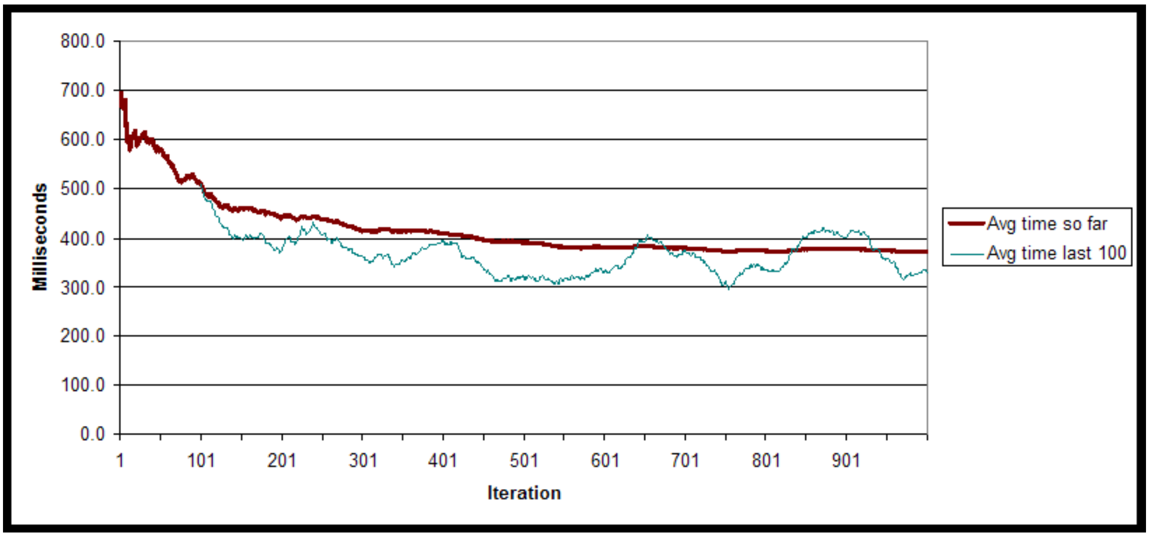


(b) Trial 2

Figure 4.13: Sample per-iteration runtime data from applying selection algorithm `LargestLevelTwoGates` to circuit C17.

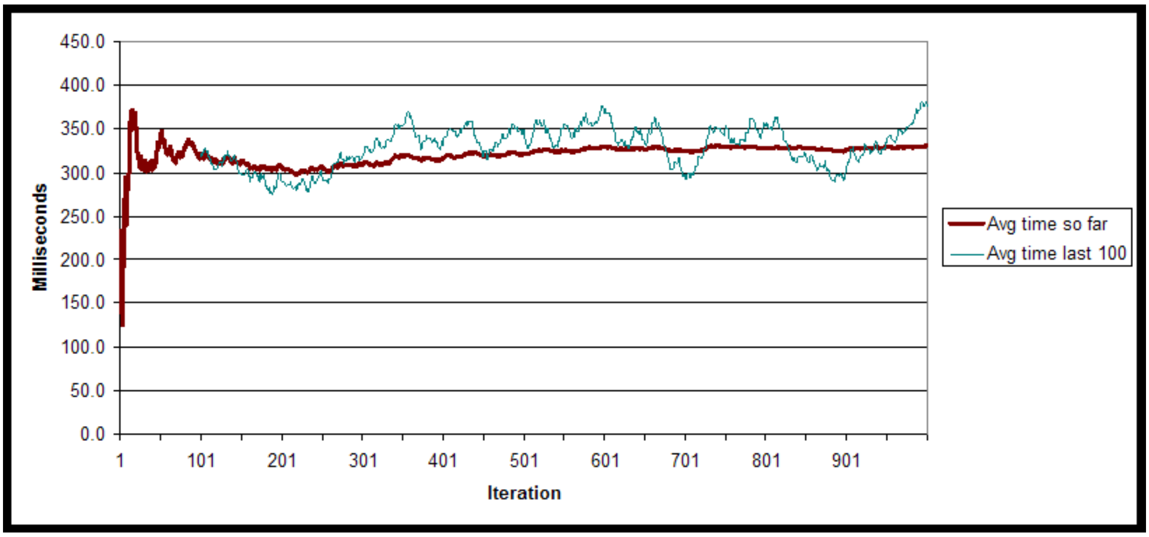


(a) Trial 1

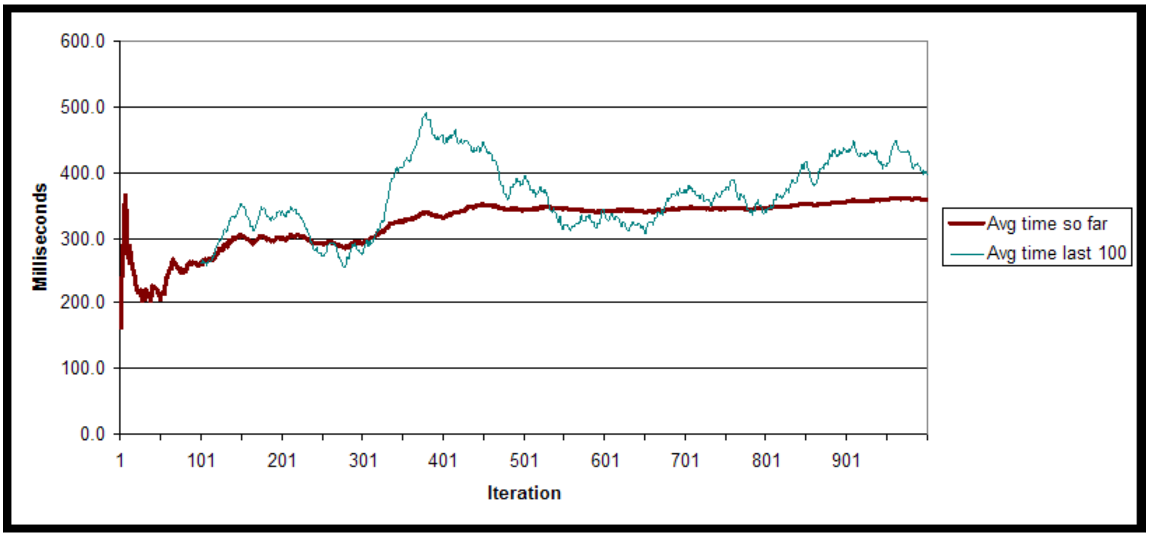


(b) Trial 2

Figure 4.14: Sample per-iteration runtime data from applying selection algorithm `LargestLevelTwoGates` to circuit C880.

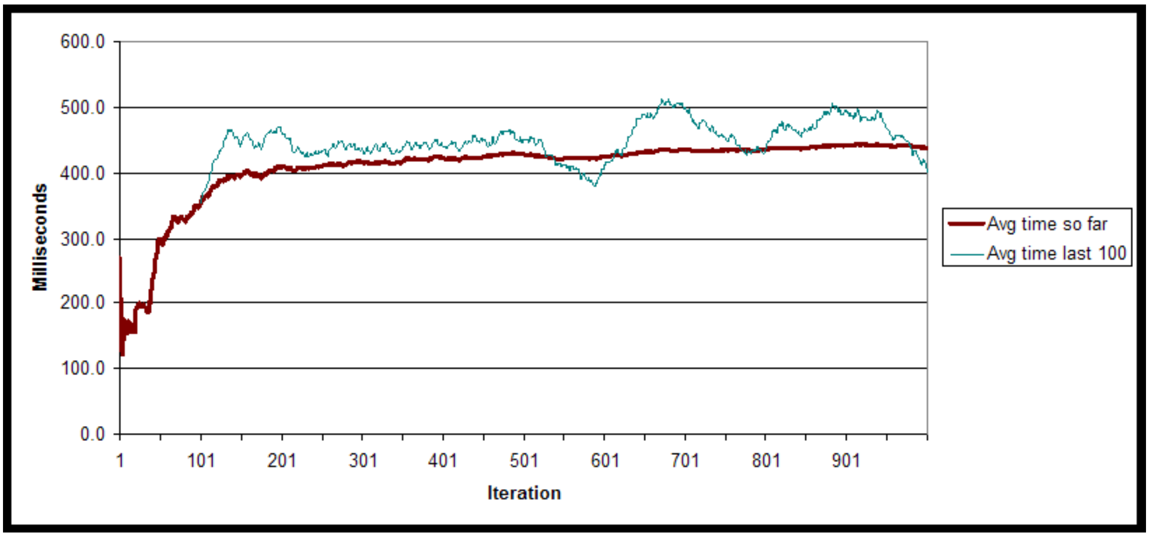


(a) Trial 1

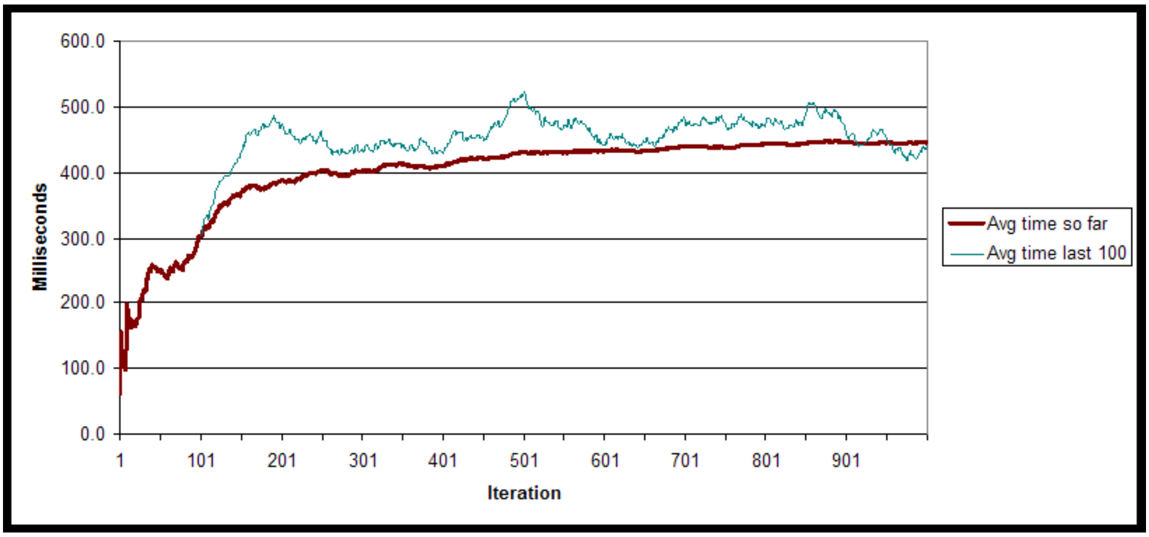


(b) Trial 2

Figure 4.15: Sample per-iteration runtime data from applying selection algorithm `OutputLevelTwoGates` to circuit C17.



(a) Trial 1



(b) Trial 2

Figure 4.16: Sample per-iteration runtime data from applying selection algorithm `OutputLevelTwoGates` to circuit C880.

V. Conclusions

The work described in the foregoing chapters comprises only the beginning of a much larger effort. Going forward, we expect a steep learning curve given the "obstacle" of the impossibility result presented in [1]. However, this research—combined with that which will follow—seeks to set *intent protection* (which alters structure and function) apart from the common understanding of *obfuscation* (which only alters structure). In this research, we focused only on the process of white-box obfuscation, a necessary but not sufficient component of program intent protection. We further narrowed our scope to white-box obfuscating combinational Boolean circuits. We developed an architecture for manipulating circuits, and developed an initial set of algorithms for white-box obfuscating circuits via subcircuit selection and replacement.

5.1 Contributions

Perhaps our biggest contribution to our area of study is CORGI, the tool upon which this and future research is based. As with any new software, its development was not without difficulty. However, without CORGI, the process of subcircuit selection and replacement would have been entirely manual which would have yielded little data: calculations by-hand would simply take too long. On the other hand, the time spent to develop a stable architecture clearly impacted the number and complexity of selection algorithms that were produced. We view this tradeoff as appropriate since it will allow future research to focus on the process of obfuscation rather than the tool that performs the task.

The six subcircuit selection algorithms we produced yielded some surprising results, and they gave us new insights into the heretofore untested process of subcircuit selection and replacement. The `RandomTwoGates` algorithm alone provided two valuable results. First, it demonstrates that the gates of a subcircuit need not be connected to be selected. Additionally, `RandomTwoGates` also demonstrates how a circuit library (CXL) can provide replacement subcircuits that introduce new control flows in the circuit. These results mean that completely disparate portions of a circuit can be intertwined, both from a black-box (functional) and a white-box (structural) perspective.

All six of the algorithms revealed that circuit size always increases when only one or two gates are selected for replacement. For single-gate subcircuits, *all* replacements have at least two gates. For a two-gate subcircuit, if its function is not semantically equivalent to a basic gate (AND, NAND, OR, NOR, XOR, or XNOR), then all replacement circuits in the circuit library will be, on average, larger than two gates. Unless and until we devise algorithms that select three or more gates can we expect to reduce circuit size. The ability to either increase or decrease circuit size is how the process of subcircuit selection and replacement will be able to produce a truly random circuit from a particular circuit family.

Finally, the three "smart" algorithms, especially, `LargestLevelTwoGates`, show how circuit growth can be controlled and predicted, even when the selection algorithm produces ever-increasing circuit size.

5.2 Future work

As alluded to above, we see 3-gate selection algorithms as the most important next step in devising an intent protection framework. One approach is to extend `RandomTwoGates` to select a third gate at random. This may be the easiest to do, but our insight is that it will provide results which will guide the development of other algorithms. In particular, as another approach, it may be advantageous during some iteration of selection to chose only subcircuits for which there is a large population of replacements in the circuit library.[1] Such a strategy will require the algorithm to find subcircuits with a particular truth table. In graph theory, this is known as subgraph isomorphism, and is an NP-complete problem. Depending on circuit size, it may nonetheless be a feasible approach.

---

[1] This assumes the library has a cache of metadata on its stores of circuit libraries which can be quickly and easily searched.

There are at least two ways CORGI can be augmented which have nothing to do with the algorithms directly. Currently, CORGI maintains no historical log of what steps and in what order were performed to obfuscate a circuit. A future version of CORGI with this capability would benefit the notion that an original circuit can be recovered from an obfuscated version. In a sense, such a log file would be analogous to a data encryption key for the white-box portion of the obfuscator. It remains to be seen what advantages might accrue for the cost of this operation, but its a question worth exploring.

Finally, CORGI is a solid proof-of-concept tool, but to make it better suited to the research, two major augmentations need to occur. An obvious shortfall is the need for a better user interface. Although not addressed in this text, the tool functionality was accessed for this research entirely through test cases since the textual user interface was too cumbersome for repeated experimentation. Ideally, a graphical user interface will be developed so that rapid selection of input parameters and selection algorithm(s) will further keep the focus on experimentation rather than coding. CORGI also needs a review of the efficiency of some of its processes (not the selection algorithms themselves). Under the hood, there are several methods which employ recursive search algorithms that are not very efficient. They become even less efficient as circuit size increases. By instituting some optimization techniques, and limiting calls to these methods only when necessary, CORGI will be more likely to achieve, at worst, polynomial slowdown for large circuits.

Appendix A. CORGI software

A.1 CORGI architecture

A.1.1 Functionality. CORGI is a Java application which employs a model-view-controller (MVC) architecture. In Figure A.1 (page 71), the *model* is the `Circuit`, which is composed of `Gate` objects. The *controller* is `CircuitController`. The *view* is the `UserCommandParser`, which provides the user a text-based user interface.

A.1.1.1 JGraphT. The Java graph library JGraphT, introduced in Section 3.3.1.1, is the "engine under the hood" of CORGI. Recall, the 'G' in CORGI stands for *graphs*, and JGraphT is what allows us to manipulate circuits as DAGs, yet elide that fact from the user. Every circuit has an underlying graph (DAG), so `Circuit` is really a façade for a JGraphT `DirectedGraph`.

All circuit modifying behavior is contained in `Circuit`; however, the mechanism of subcircuit selection and replacement is modularized as a separate class, ... (more to come)

A.2 Non-selection algorithms

For the sake of brevity in the main text, the discussion of the non-selection algorithms is presented here. The entire process of subcircuit selection and replacement is given in Algorithm 7. The procedures for `removeSubCircuit`, `fetchReplacement`, and `insertReplacement` are elided since they are purely "mechanical" in the sense that they do not impact the selection process. Once a subcircuit C_{sub} is selected from circuit C'_i, then these three methods will, respectively, remove C_{sub}, get a replacement circuit C_{rep} from CXL, then insert C_{rep} in place of C_{sub}.

Algorithm 8 (`SelectRandomGate`) and Algorithm 9 (`RejectGates`) are helper methods used by the six selection algorithms discussed in Chapter IV. `SelectRandomGate` simply selects a single gate at random from among a set of gates. This capability is needed since subcircuit selection relies on a sequence of random gate selections.

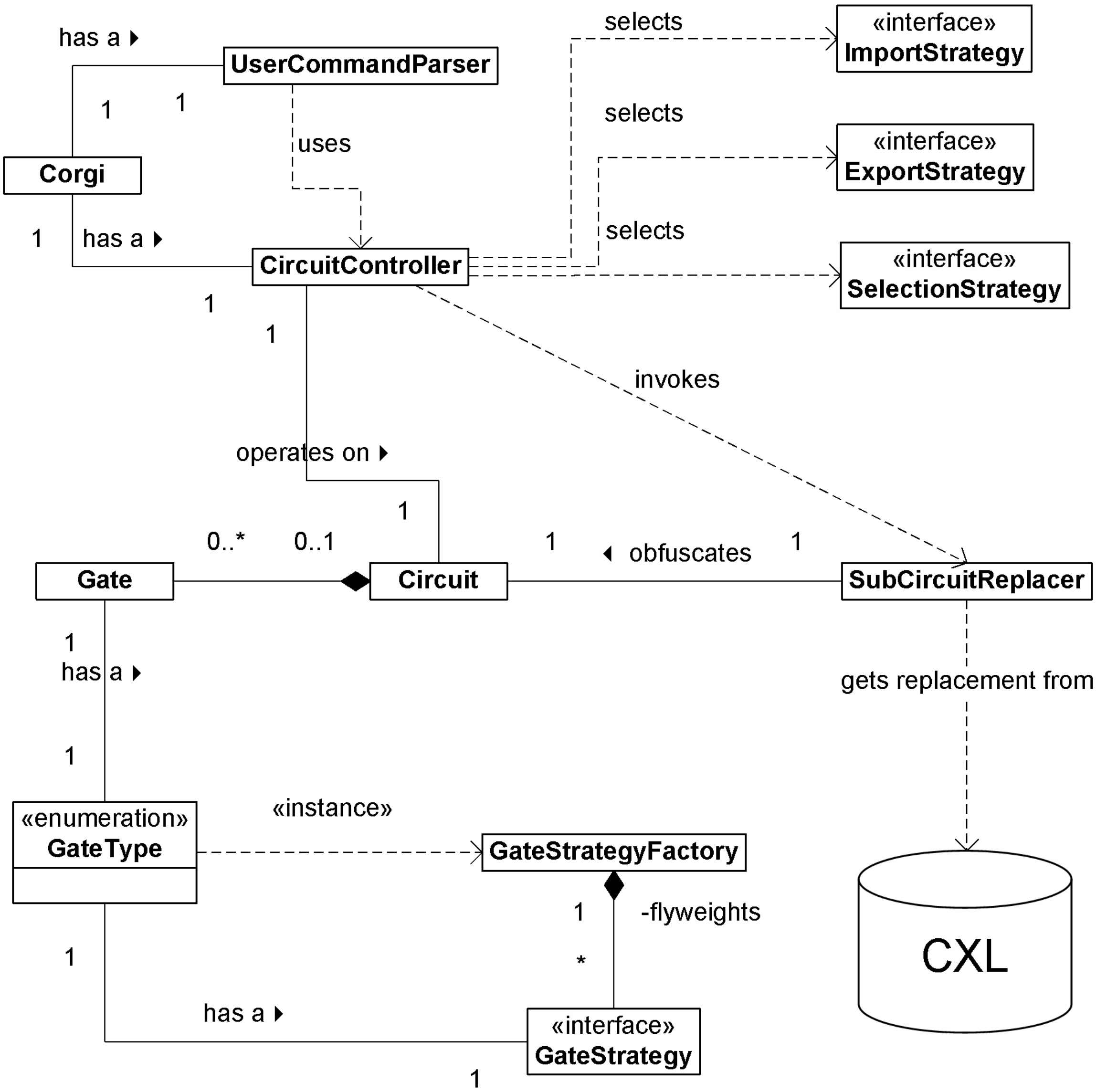


Figure A.1: The UML class diagram which shows the CORGI architecture.

Algorithm 7 `performReplacement(Selection(`C'_i`))`

```
$C'_i \leftarrow$ circuit $C$ after $i$ iterations of randomization
$G_{sub} \leftarrow \varnothing$ {subset of gates in $C'_i$: $G_{sub} \subset G(C'_i)$}
$G_{sub} \leftarrow$ call Selection($C'_i$) {the interface for the selection algorithms}
$C_{sub} \leftarrow$ call RemoveSubCircuit($G_{sub}$)
$C_{rep} \leftarrow$ call FetchReplacement($C_{sub}$) {this is the CXL interface}
$C'_{i+1} \leftarrow$ call InsertReplacement($C_{rep}$)
return $C'_{i+1}$ {circuit $C'_i$ after replacing $C_{sub}$ with $C_{rep}$}
```

Algorithm 8 `SelectRandomGate(`G`)`

```
Require: $G$ is a non-empty set of gates
$k \leftarrow$ uniform random number such that $0 \leq k < |G|$
$g_k \leftarrow$ the $k^{\text{th}}$ gate in $G$
return $g_k$
```

`RejectGates` identifies the set of all gates which lie on all paths through a particular gate and which are more than one hierarchical level removed from said gate. `RejectGates` is the means by which `performReplacement` prevents cycles from being introduced in C'_{i+1} when replacing a subcircuit that contains more than one gate.

Algorithm 9 `RejectGates`(g_k, P)

```
Require: $P$ true for predecessors of $g_k$, false for successors of $g_k$
$G_{rej} \leftarrow \varnothing$ {set of rejected gates}
$G_{curr} \leftarrow \varnothing$ {set of gates being considered for rejection}
$G_{prev} \leftarrow \varnothing$ {set of gates already considered for rejection}
$G_{next} \leftarrow \varnothing$ {set of gates to be considered for rejection}
$G_{adj} \leftarrow \varnothing$ {set of predecessors (successors) of a gate}
$G_{curr} \leftarrow G_{curr} + g_k$
if $P =$ true then
    $G_{adj} \leftarrow$ predecessors of $g_k$
else
    $G_{adj} \leftarrow$ successors of $g_k$
end if
for all gates $g_a$ in $G_{adj}$ do
    if difference between hierarchy levels of $g_a$ and $g_k is > 1$ then
        $G_{curr} \leftarrow G_{curr} \cup \{g_a\}$
    end if
end for
$G_{adj} \leftarrow \varnothing$
while $G_{curr} \neq \varnothing$ do
    $G_{next} \leftarrow \varnothing$
    for all gates $G_c$ in $G_{curr}$ do
        if $P ==$ true then
            $G_{adj} \leftarrow$ predecessors of $G_c$
        else
            $G_{adj} \leftarrow$ successors of $G_c$
        end if
        $G_{next} \leftarrow G_{next} \cup G_{adj}$
    end for
    $G_{prev} \leftarrow G_{prev} \cup G_{curr}$
    $G_{curr} \leftarrow \varnothing$
    $G_{curr} \leftarrow G_{curr} \cup G_{next}$
end while
return $G_{rej}$
```

`EstablishGateHierarchy` is a circuit function that sets the `hierarchy` attribute for all gates in the circuit. When there are multiple paths between a particular pair of gates, and when one path is shorter than the other (in terms of number of gates along the path), then one or more of the gates on the shorter path could legally occupy any one of several levels in the hierarchy. We choose to assign gates to the lowest possible level that adheres to this convention: *every* gate in the circuit will *always* occupy a level that is lower (smaller) than the level of *any* of its predecessors.

Algorithm 10 `EstablishGateHierarchy()`

```
label all gates as $\ell_0$
$\ell_G \leftarrow 0$ {initialize global maximum level}
$\ell_L \leftarrow 0$ {initialize local (output) maximum level}
for all circuit output gates $g_{out}$ do
    $\ell_L \leftarrow$ call SetGateHierarchies($g_{out}, 0, 0$)
    $\ell_G \leftarrow$ MAX($\ell_L, \ell_G$)
end for
```

None of the so-called *level*-based selection algorithms would function properly without `EstablishGateHierarchy`. `EstablishGateHierarchy`, in turn, relies upon the recursive function `SetGateHierarchies` (described in Algorithm 11). The way it works is to perform a DFS beginning at each circuit output, explore that output's predecessor tree (in the underlying DAG), and set the the `hierarchy` attribute for all gates along the way. Some pruning is performed, but there will invariably be gates that are visited at least twice, which makes `EstablishGateHierarchy` inefficient. Since so much of CORGI relies on gates having a correct `hierarchy` attribute, future versions of CORGI will benefit greatly from optimizing `EstablishGateHierarchy`.

Algorithm 11 `SetGateHierarchies`(g_i, ℓ_L, ℓ_G)

```
for all predecessor gates $g_j$ of gate $g_i$ do
    if $\ell(g_j) \leq \ell(g_i)$ then
        $\ell(g_j) \leftarrow \ell(g_j) + 1$
        $\ell_G \leftarrow$ call SetGateHierarchies($g_j, \ell_L + 1, \ell_G$)
    end if
    return MAX($\ell_L, \ell_G$)
end for
```

A.3 Selection algorithm behavior

Figures A.2, A.3, and A.4 give insight into the behavior of the six selection algorithms.

Algorithm	H_{max}	H_{avg}	H_{min}	H_{σ}
R1G	291	183.7	117	61.5
OL2G	103	97.7	90	4.8
R2G	119	89.9	75	15.6
FL2G	78	69.6	62	5.1
RL2G	87	65.6	46	14.0
LL2G	46	35.2	31	4.2

(a)

Algorithm	W_{max}	W_{avg}	W_{min}	W_{σ}
R1G	9	6.3	4	1.5
OL2G	5	4.4	4	0.5
R2G	7	5.3	4	1.1
FL2G	6	5.4	5	0.5
RL2G	8	6.8	5	1.2
LL2G	20	14.8	12	2.4

(b)

Algorithm	H_{avg}/W_{avg}	Growth(%)
R1G	29.2	90.4
OL2G	22.2	47.4
R2G	17.0	43.5
FL2G	12.9	33.3
RL2G	9.6	31.3
LL2G	2.4	16.1

(c)

Figure A.2: Experimental results from performing ten trials of 200 iterations each using all six selection algorithms, with ISCAS circuit C17 as the target C. To provide a common mode of comparison, all three tables are sorted in decreasing order of H_{avg}.
(a) The number of hierarchical levels in C' (maximum, average, minimum, and standard deviation).
(b) The number of gates in the widest hierarchical level of C' (maximum, average, minimum, and standard deviation).
(c) Height-to-width ratio and rate at which number of hierarchy levels increase per iteration.

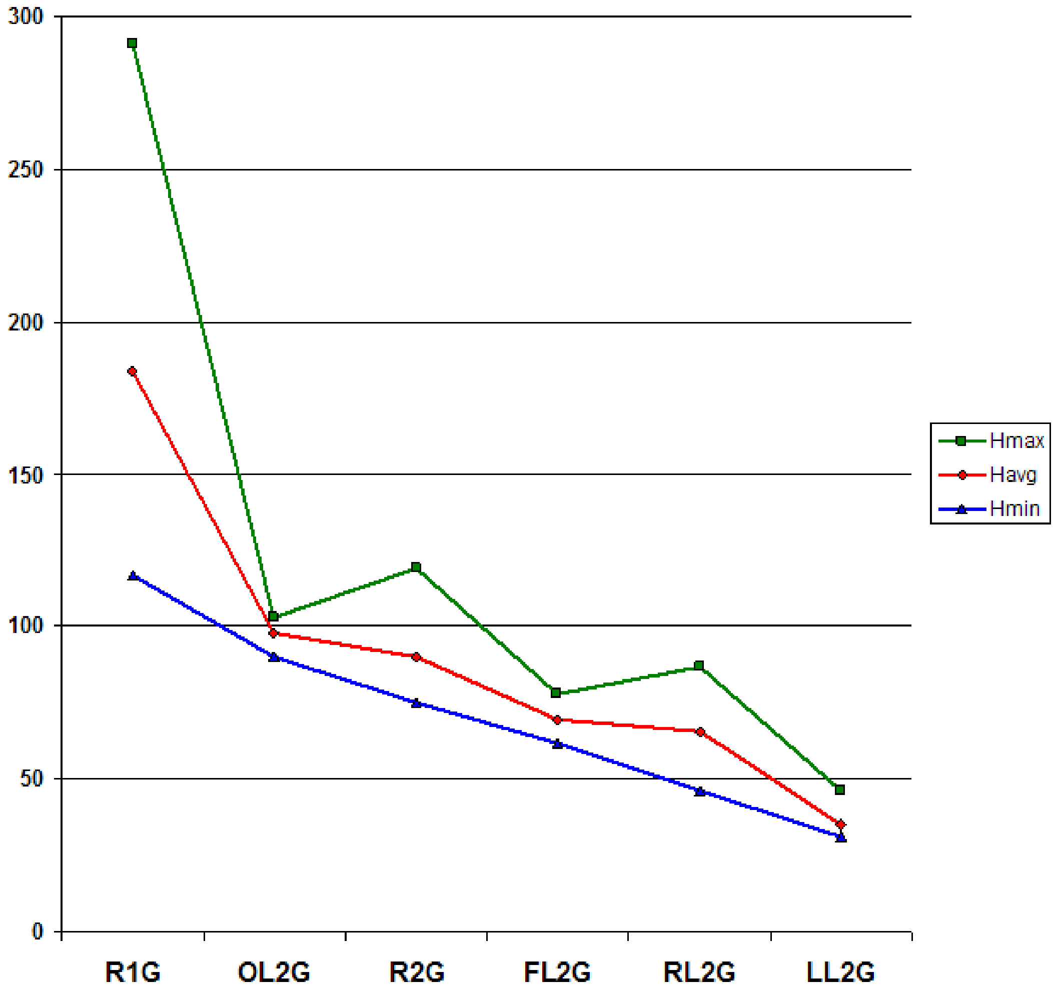


Figure A.3: Chart of data from Figure A.2(a).

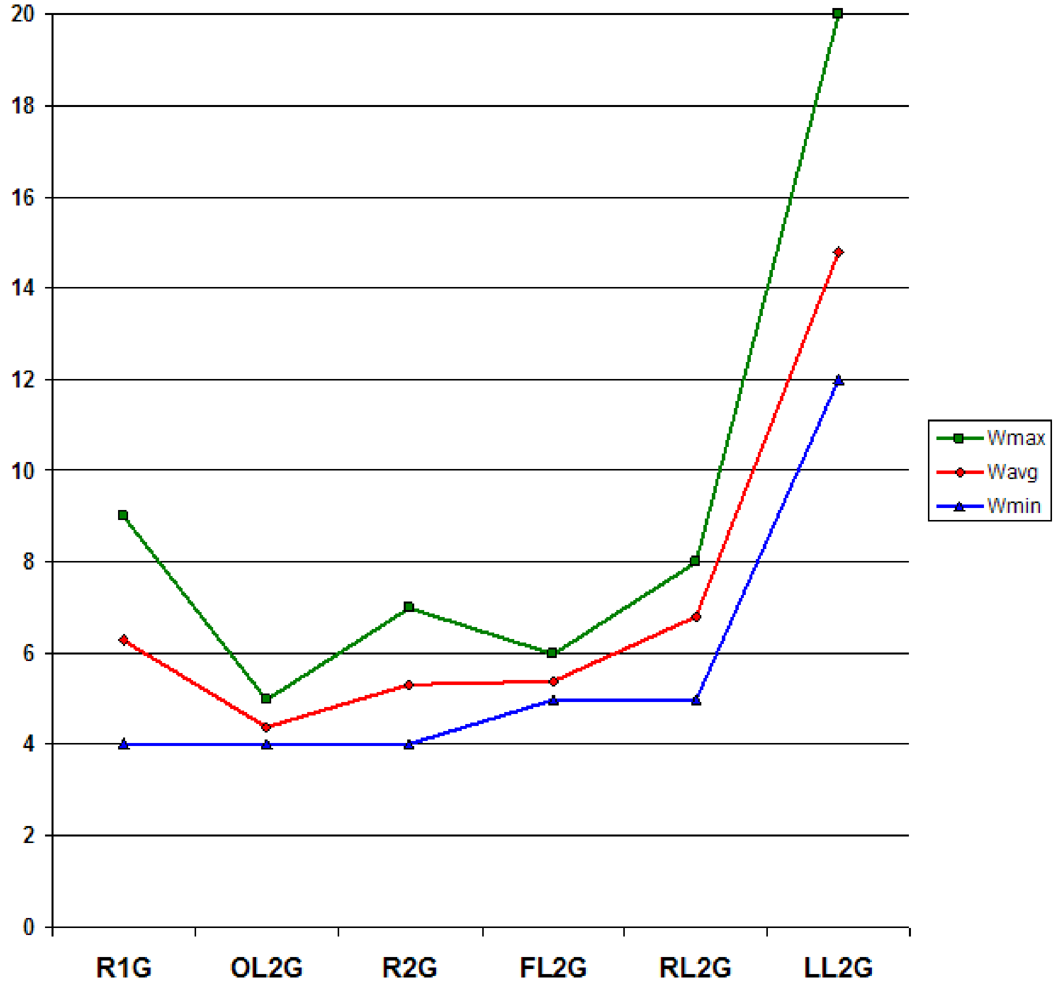


Figure A.4: Chart of data from Figure A.2(b).

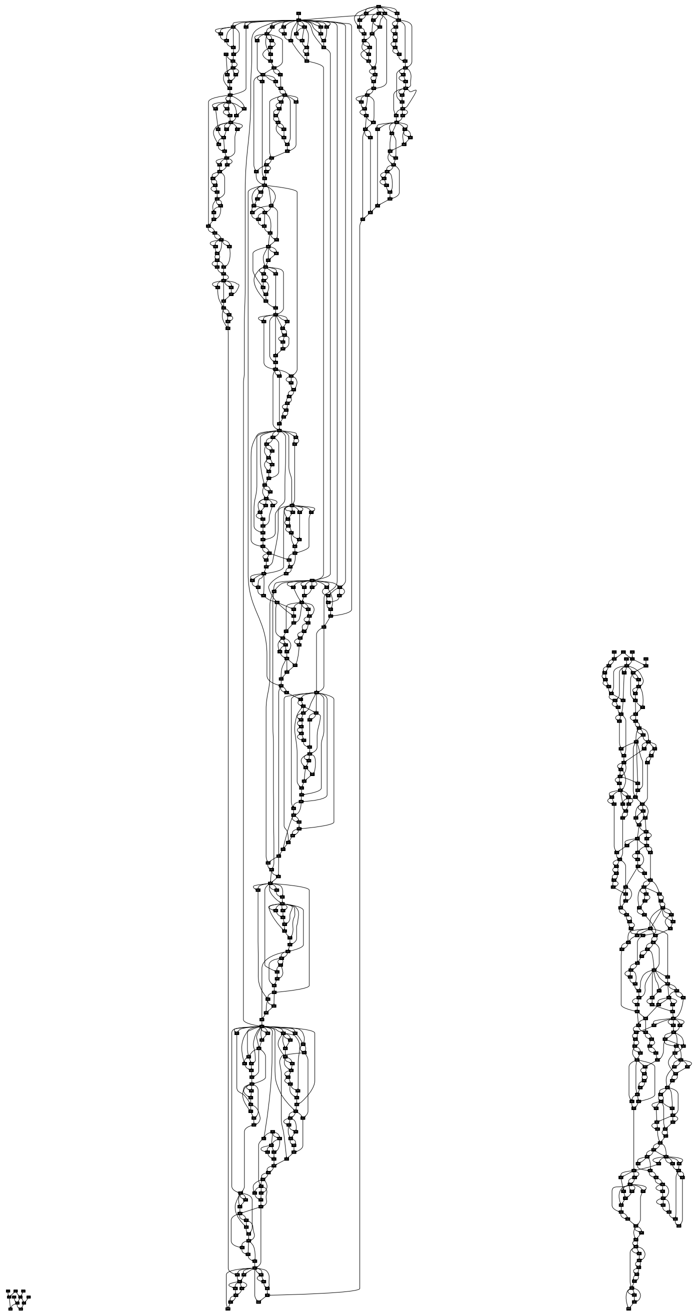
A.4 Selection algorithm results

A.4.1 C17 with all algorithms. Figures A.5, A.6, and A.7 display examples of the results achieved when each of the six algorithms are applied to a simple ISCAS benchmark circuit, C17. In each case, the algorithm ran for 200 iterations. The images are DAGs which represent the various circuits. While they are not strictly circuits, they demonstrate the behavior of each algorithm. All images are drawn to relative scale for ease of comparison.[1]

A.4.2 C880 with `OutputLevelTwoGates`. Figures A.8, A.9, and A.10 shows how circuit C880 changes over time when randomized using the `OutputLevelTwoGates` selection algorithm. Compare Figure A.8 to Figure A.5(c). Note that C880, which has 26 outputs, grows in height much more slowly than does C17, which has 2 outputs, when `OutputLevelTwoGates` is applied for 200 iterations.

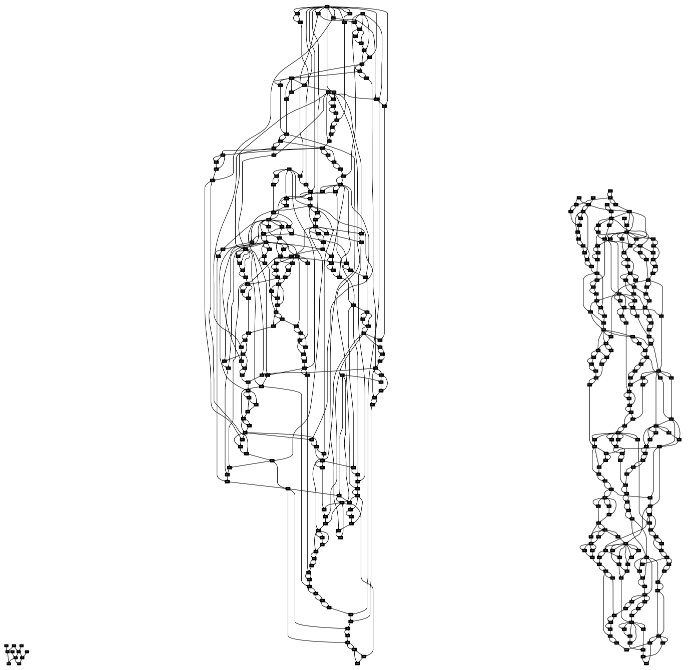
---

[1] When viewing this document electronically in PDF format, the circuit details can be seen by zooming in to at least 1600% magnification.



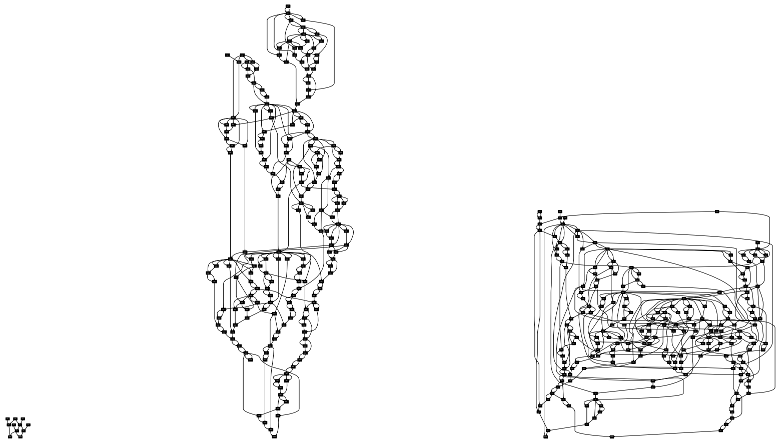
(a) (b) (c)

Figure A.5: Comparison of original circuit (ISCAS C17) to sample results of R1G and OL2G algorithms (200 iterations; circuits represented as DAGs).
(a) C = ISCAS benchmark circuit C17 (height = 3 levels, width = 3 gates).
(b) C' after applying `RandomSingleGate` to C (height = 189 levels, width = 7 gates).
(c) C' after applying `OutputLevelTwoGates` to C (height = 93 levels, width = 4 gates).



(a) (b) (c)

Figure A.6: Comparison of original circuit (ISCAS C17) to sample results of R2G and FL2G algorithms (200 iterations; circuits represented as DAGs).
(a) C = ISCAS benchmark circuit C17 (height = 3 levels, width = 3 gates).
(b) C' after applying `RandomTwoGates` to C (height = 93 levels, width = 6 gates).
(c) C' after applying `FixedLevelTwoGates` to C (height = 67 levels, width = 6 gates).



(a) (b) (c)

Figure A.7: Comparison of original circuit (ISCAS C17) to sample results of RL2G and LL2G algorithms (200 iterations; circuits represented as DAGs).
(a) C = ISCAS benchmark circuit C17 (height = 3 levels, width = 3 gates).
(b) C' after applying `RandomLevelTwoGates` to C (height = 61 levels, width = 8 gates).
(c) C' after applying `LargestLevelTwoGates` to C (height = 32 levels, width = 15 gates).

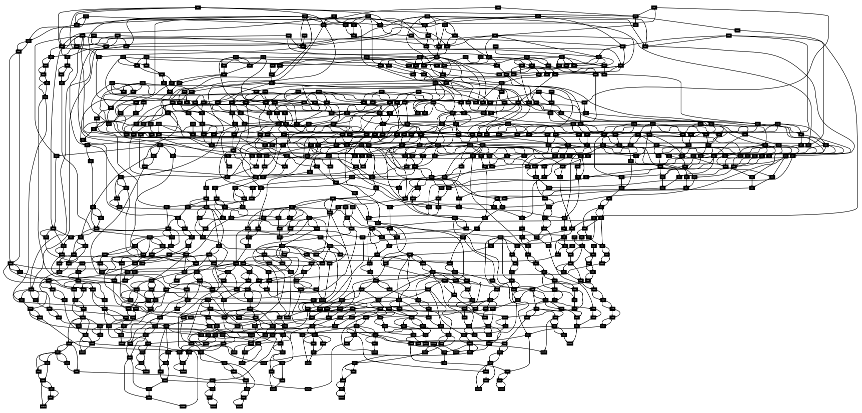


Figure A.8: C' after applying 200 iterations of `OutputLevelTwoGates` to ISCAS benchmark circuit C880 (height= 42 levels, width= 38 gates).

Figure A.9: C' after applying 400 iterations of `OutputLevelTwoGates` to ISCAS benchmark circuit C880 (height= 60 levels, width= 31 gates).

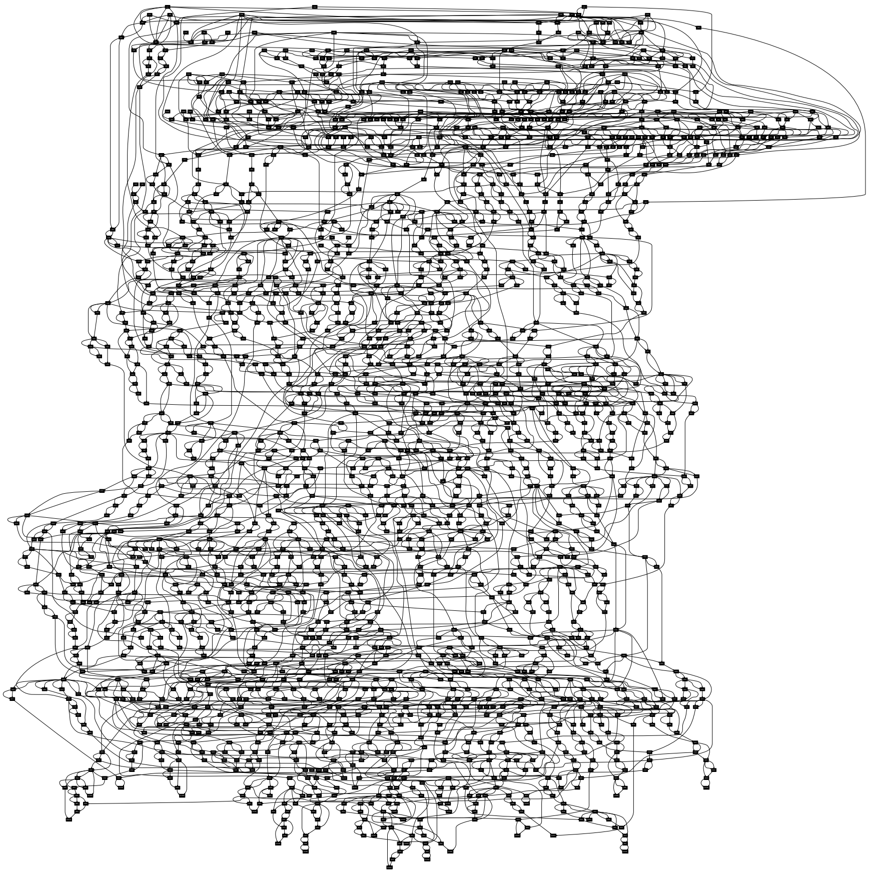


Figure A.10: C' after applying an *additional* 800 iterations of `OutputLevelTwoGates` to the circuit C' in Figure A.9 (height= 98 levels, width= 33 gates).

Bibliography

1. Barak, Boaz, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. "On the (Im)possibility of obfuscating programs". *Electronic Colloquium on Computational Complextiy*, 8(57):1–41, 2001.
2. "Benchmark circuits". Internet: `http://www.fm.vslib.cz/~kes/asic/iscas/`, Jan 2007.
3. Collberg, Christian, Clark Thomborson, and Douglas Low. *A Taxonomy of Obfuscating Transformations.* Technical Report 148, University of Auckland, Jul 1997. URL `http://www.cs.arizona.edu/~collberg/Research/Publications/`.
4. Edwards, Stephen A. "Making cyclic circuits acyclic". *DAC '03: Proceedings of the 40th conference on Design automation*, 159–162. ACM, New York, NY, USA, 2003. ISBN 1-58113-688-9.
5. Garey, M. R. and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* W. H. Freeman, January 1979. ISBN 0-716-71045-5.
6. Goldwasser, Shafi and Guy N. Rothblum. "On Best-Possible Obfuscation". *4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, 194–213. Springer, 21-24 February 2007. ISBN 3-540-70935-5.
7. Gross, Jonathan L. and Jay Yellen. *Graph Theory and its Applications.* Chapman & Hall/CRC, 2 edition, 2006. ISBN 1-58488-505-X.
8. Hansen, Mark C., Hakan Yalcin, and John P. Hayes. "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering". *IEEE Des. Test*, 16(3):72–80, 1999. ISSN 0740-7475.
9. Huth, Michael and Mark Ryan. *Logic in computer science: Modelling and Reasoning about Systems.* Cambridge University Press, 2004.
10. James, Moses C. *Obfuscation Framework Based on Functionally Equivalent Combinatorial Logic Families.* Master's thesis, Air Force Institute of Technology, WPAFB, OH, March 2008.
11. Kukis, Mark and Katherine Arms. "Bush to China: Return Plane, Crew". Internet: `http://www.military.com/Content/MoreContent1?file=standoff`, April 2001.
12. McDonald, Jeffrey T. *Enhanced Security for Mobile Agent Systems.* Ph.D. thesis, Florida State University, 2006.
13. Mish, Frederick C. (editor). *Merriam-Webster's collegiate dictionary.* Merriam-Webster, Incorporated, Springfield, MA, 10 edition, 2001. ISBN 0-87779-710-2.
14. Naveh, Barak. "JGraphT". Internet: `http://jgrapht.sourceforge.net/`, January 2008.

15. “PreEmptive Solutions”. Internet: `http://www.preemptive.com/`, Jan 2008.

16. “Semantic Designs, Inc.” Internet:
`http://www.semdesigns.com/Products/Obfuscators/`, Jan 2008.

17. “Smardec”. Internet: `http://www.smardec.com/products.html`, Jan 2008.

18. Varnovsky, Nikolay P. and Vladimir A. Zakharov. “On the Possibility of Provably Secure Obfuscating Programs.” Manfred Broy and Alexandre V. Zamulin (editors), *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, 91–102. Springer, 2003. ISBN 3-540-20813-5.

19. “Wiktionary”. Internet: `http://en.wiktionary.org/`, Oct 2007.

Vita

Major Kenneth E. Norman graduated from Fayette County High School in Fayetteville, Georgia. He entered undergraduate studies at the Georgia Institute of Technology in Atlanta, Georgia where he graduated with a Bachelor degree in Electrical Engineering in 1992. He was commissioned through Officer Training School in 1993. In 2002, he earned his first Masters degree in Engineering Management at the Florida Institute of Technology.

Major Norman was first assigned to HQ Standard Systems Group, Maxwell AFB, Alabama in July 1993 as officer in charge of software development. In October 1996, he was assigned to the National Air Intelligence Center, Wright-Patterson AFB, Ohio where he served as an intelligence analyst. His third assignment began in October 1999 when he was selected to stand up a new joint interoperability program office at the US Army's Communications-Electronics Command, Fort Monmouth, New Jersey. Next, he became an assignment officer for the developmental engineer career field at HQ Air Force Personnel Center, Randolph AFB, Texas in July 2002. Maj Norman was next assigned to the National Reconnaissance Office in Chantilly, Virginia in August 2004 as a systems engineer. While there, he was selected for in-residence Intermediate Developmental Education, which precipitated his assignment to attend the Air Force Institute of Technology in August 2006. Upon graduation, he will remain at Wright-Patterson AFB for his assignment to Air Force Research Laboratory.

Permanent address: Air Force Institute of Technology
2950 Hobson Way
Wright-Patterson AFB, OH 45433-7765

Index

The index is conceptual and does not designate every occurrence of a keyword. Page numbers in bold represent concept definition or introduction.

acyclic graph, **17**

black-box obfuscation, 8

circuit class, 13
CORGI, 20
CXL, 20

digraph, *see* directed graph
directed acyclic graph, **17**
directed graph, **15**

graph, **15**

multi-graph, **17**

Random Program Model, 8
RPM, *see* Random Program Model

subcircuit replacement, 18
subcircuit selection, 18

VBB, *see* virtual black box
virtual black box, 4

white-box obfuscation, 8

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)* 27-03-2008	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sep 2006–Mar 2008

4. TITLE AND SUBTITLE

Algorithms for White-box Obfuscation Using
Randomized Subcircuit Selection and Replacement

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Norman, Kenneth E., Maj, USAF

5d. PROJECT NUMBER

08-183

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/08-17

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research
801 North Randolph Street, Rm 732
Arlington VA 22203-1977
703–696–9544 (DSN: 426)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Software protection remains an active research area with the goal of preventing adversarial software exploitation such as reverse engineering, tampering, and piracy. Heuristic obfuscation techniques lack strong theoretical underpinnings while current theoretical research highlights the impossibility of creating general, efficient, and information theoretically secure obfuscators.

In this research, we consider a bridge between these two worlds by examining obfuscators based on the Random Program Model (RPM). Such a model envisions the use of program encryption techniques which change the black-box (semantic) and white-box (structural) representations of underlying programs. In this thesis we explore the possibilities for white-box transformation. Under an RPM formulation, if an adversary cannot distinguish an original program from either its obfuscated version (whose black-box behavior has been strategically altered) or a randomly generated program of comparable size, then the white-box intent of the original program has been sufficiently protected. One proposed method of creating such random indistinguishability is by choosing (at random) a program from a size-bounded set of all semantically equivalent possibilities.

Since full enumeration of reasonably sized programs is not possible, in this work we focus on obfuscators which introduce random white-box structural variation based on iterative selection and replacement. We design and develop an obfuscation framework for programmatic logic expressed as combinatorial Boolean circuits and compare six unique approaches for sub-circuit selection. We analyze the relative behavior of random and guided-random sub-circuit selection algorithms while showing their utility in producing random white-box structural variation.

15. SUBJECT TERMS

software tools, software engineering, computer programs, cryptography, obscuration, software obfuscation, randomization, pseudo random sequences, random functions

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Lt Col J. Todd McDonald
U	U	U	UU	99	**19b. TELEPHONE NUMBER** *(include area code)* 937–255–3636 x4639, jmcdonal@afit.edu